ZED SHAW'S HARD WAY SERIES



Learn C the HARD WAY

Practical Exercises on the Computational Subjects You Keep Avoiding (Like C)

ZED A. SHAW

目錄

笨办法学C 中文版

来源：Learn C The Hard Way

作者：Zed A. Shaw

译者：飞龙

自豪地采用谷歌翻译

> 一句 **MMP** 送给在座的各位程序正义垃圾。

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- Github

赞助我



协议

前言

> 原文：Preface
>
> 译者：飞龙

这是本书创作中的转储版本，所用的措辞可能不是很好，也可能缺失了一些章节，但是你可以看到我编写这本书的过程，以及我的做事风格。

你也可以随时发送邮件到help@learncodethehardway.org来向我寻求帮助，我通常会在1~2天之内答复。

这个列表是一个讨论列表，并不只允许发布公告，它用于讨论本书和询问问题。

最后，不要忘了我之前写过笨办法学Python，如果你还不会编程，你应该先读完它。LCTHW并不面向初学者，而是面向至少读完LPTHW或者已经懂得一门其它编程语言的人。

常见问题

这门课程需要多少时间？

你应该花一些时间直到你掌握它，并且每天都要坚持编写代码。一些人花了大约三个月，其它人花了六个月，还有一些人只用了一个星期。

我需要准备什么样的电脑？

你需要OSX或者Linux来完成这本书。

导言：C的笛卡尔之梦

> 原文：Introduction: The Cartesian Dream Of C
>
> 译者：飞龙

> 直到现在，凡是我当作最真实、最可靠而接受的东西，都是从感官或通过感官得来的。不过，我有时觉得这些感官是骗人的，并且为了小心谨慎起见，对于一经骗过我们的东西就决不完全加以信任。
>
> 勒内·笛卡尔，《第一哲学沉思录》

如果有一段引述用来描述C语言编程的话，那就是它了。对于大多数程序员，C是极其可怕而且邪恶的。他就像是恶魔、撒旦，或者一个使用指针的花言巧语和对机器的直接访问来破坏你生产力的骗子洛基。于是，一旦这位计算界的路西法将你缠住，他就会使用邪恶的“段错误”来毁掉你的世界，并且揭露出与你交易中的骗局而嘲笑你。

然而，C并不应由于这些事实而受到责备。你的电脑和控制它的操作系统才是真正的骗子，而不是朋友。它们通过密谋来向你隐藏它们的真实执行逻辑，使你永远都不真正知道背后发生了什么。C编程语言的失败之处只是向你提供接触背后真正工作原理的途径，并且告诉了你一些难以接受的事实。C会向你展示痛苦的真像（红色药丸），它将幕布拉开来向你展示一些神奇的原理。C即是真理。

既然C如此危险，为什么还要使用它？因为C给了你力量来穿越抽象的假象，并且将你从愚昧中解放出来。

你会学到什么

这本书的目的是让你足够熟悉C语言，并能够使用它编写自己的软件，或者修改其他人的代码。这本书的最后，我们会从一本叫做“K&R C”的名著中选取实际的代码，并且用你学过的知识来做代码审查。你需要学习下面这些东西来达到这一阶段：

+ C的基本语法和编写习惯。
+ 编译，`make`文件和链接。
+ 寻找和预防bug。
+ 防御性编程实践。
+ 使C的代码崩溃。
+ 编写基本的Unix系统软件。

截至最后一章，你将会有足够的工具来解决基本的系统软件、库和其它小项目。

如何阅读本书

这本书为那些已经掌握至少一门编程语言的人而设计。如果你还没有接触过编程，我推荐你先学习笨办法学Python，这本书适用于真正的新手并且适合作为第一本编程书。一旦你学会了Python，你可以返回来开始学习这本书。

对于那些已经学会编程的人，这本书的开头可能有些奇怪。它不像其它书一样，那些书中你会阅读一段段的文字然后编写一些代码。相反，这本书中我会让你立即开始编程，之后我会解释你做了什么。这样更有效果，因为你已经经历过的事情解释起来更加容易。

由于采用了这样的结构，下面是本书中你必须遵守的规则：

- 手动输入所有代码。不要复制粘贴！
- 正确地输入所有代码，也包括注释。
- 运行代码并保证产生相同的输出。
- 如果出现了bug则修正它。
- 做附加题时，如果你做不出某道题，马上跳过。
- 在寻求帮助之前首先试着自己弄懂。

如果你遵守了这些规则，完成了本书的每一件事，并且还不会编程C代码的话，你至少尝试过了。它并不适用于每个人，但是尝试的过程会让你成为一个更好的程序员。

核心能力

我假设你之前使用为“弱者”设计的语言。这些“易用的”语言之一是Python或者Ruby，它们带给了你草率的思维和半吊子的黑魔法。或者，你可能使用类似Lisp的语言，它假设计算机是纯函数式的奇幻大陆，带有一些为婴儿准备的充气墙。再或者你可能学过Prolog，于是你认为整个世界都是一个数据库，你可以从中寻找线索。甚至更糟糕的是，我假设你一直都在用IDE，所以你的大脑布满了内存漏洞，并且你每打三个字符都要按CTRL+空格来打出函数的整个名字。

无论你的背景如何，你都可能不擅长下面四个技能：

阅读和编写

如果你使用IDE这会尤其正确。但是总体上我发现程序员做了很多“略读”，并且在理解上存在问题。它们会略读需要详细理解的代码，并且觉得他们已经理解了但事实上没有。其它语言提供了可以让他们避免实际编写任何代码的工具，所以面对一种类似C的语言时，他们就玩完了。你需要知道每个人都有这个问题，并且你可以通过强迫自己慢下来并且仔细对待阅读和编写代码来改正它。一开始你可能感到痛苦和无聊，但是这样的次数多了它也就变得容易了。

专注细节

每个人都不擅长这方面，它也是劣质软件的罪魁祸首。其它语言让你不会集中注意力，但是C要求你集中全部注意力，因为它直接在机器上运行，并且机器比较挑剔。C中没有“相似的类型”或者“足够接近”，所以你需要注意，再三检查你的代码，并假设你写的任何代码都是错的，直到你能证明它是对的。

定位差异

其它语言程序员的一个关键问题就是他们的大脑被训练来指出那个语言的差异，而不是C。当你对比你的代码和我练习中的代码时，你的眼睛会跳过你认为不重要或者不熟悉的字符。我会给你一些策略来强制你观察你的错误，但是要记住如果你的代码并不完全像书中的代码，它就是错的。

规划和调试

我喜欢其它较简单的语言，因为我可以想怎么写就怎么写。我将已有的想法输入进解释器，然后可以立即看到结果。你可以把你的想法试验出来，但是要注意，如果你仍然打算“试验代码使其能够工作”，它就行不通了。C对于你来说稍困难，因为你需要规划好首先创建什么。的确，你也可以进行试验，但是比起其他语言，你必须在C中更早地严肃对待代码。我会教给你在编程之前规划程序核心部分的方法，这对于使你成为更好的程序员十分有帮助。即使一个很小的规划，都会使接下来的事情变得顺利。

学习C语言会使你变成更好的程序员，因为会强制你更早、更频繁地解决这些问题。你不会再草率地编写半吊子的代码，代码也会能够正常工作。C的优势是，它是一个简单的语言，你可以自己来弄清楚，这使得它成为用于学习机器，以及提升程序员核心技能的最佳语言。

C比其它语言都要难，而这是由于C并不对你隐藏细节，它们在其它语言中都试图并且未能被掩盖。

协议

练习0：准备

> 原文：Exercise 0: The Setup
>
> 译者：飞龙

在这一章中，你将为C语言编程配置好你的系统。一个好消息是对于所有使用Linux或者Mac的人，你的系统是为C语言编程而设计的。C语言的创造者也对Unix操作系统的创造做出了贡献，并且Linux和OSX都是基于Unix的。事实上，安装工作会非常简单。

对于Windows上的用户，我有一个坏消息：在Windows上学习C非常痛苦。你可以在Windows上编写C代码，这并不是问题。问题是所有的库、函数和工具都和其它的C语言环境有些差异。C来自于Unix，并且和Unix平台配合得比较好。恐怕这是一个你并不能接受的事实。

然而你并不需要为此恐慌。我并不是说要完全避免Windows。然而我说的是，如果你打算以最短的时间来学习C，你需要接触Unix并适应它。这同时也对你有帮助，因为懂得一些Unix的知识，也会让你懂得一些C编程的习惯，以及扩充你的技能。

这也意味着每个人都需要使用命令行。嗯，就是这样。你将会进入命令行并且键入一些命令。不要为此感到害怕，因为我会告诉你要键入什么，以及结果应该是什么样子，所以你实际上会学到很多东西，同时扩充自己的技能。

Linux

在多数Linux系统上你都需要安装一些包。对于基于Debian的系统，例如Ubuntu你需要使用下列命令来安装一些东西：

```
$ sudo apt-get install build-essential
```

上面是命令行提示符的一个示例。你需要接触到能输入它的地方，找到你的“终端”程序并且运行它。接着，你会看到一个类似于 `$` 的Shell提示符，并且你可以在里面键入命令。不要键入 `$` ，而是它后面的东西。

下面是在基于RPM的Linux系统，例如Fedora中执行相同安装工作的方法：

```
$ su -c "yum groupinstall development-tools"
```

一旦你运行了它，它会正常工作，你应该能够做本书的第一个练习。如果不能请告诉我。

Mac OSX

在 Mac OSX上，安装工作会更简单。首先，你需要从苹果官网下载最新的 `XCode` ，或者找到你的安装DVD并从中安装。需要下载的文件很大，要花费很长时间，所以我推荐你从DVD安装。同时，上网搜索“安装xcode”来指导你来安装它。

一旦你安装完XCode，可能需要重启你的电脑。你可以找到你的终端程序并且将它放到快捷启动栏中。在本书中你会经常用到终端，所以最好将它放到顺手的区域。

Windows

对于Windows用户，你需要在虚拟机中安装并运行一个基本的Ubuntu Linux系统，来做本书的练习，并且避免任何Windows中安装的问题。

> 译者注：如果你的Windows版本是Win10 14316及之后的版本，可以开启Ubuntu子系统来获取Linux环境。

文本编辑器

对于程序员来说，文本编辑器的选择有些困难。对于初学者我推荐他们使用 `Gedit` ，因为它很简单，并且可以用于编写代码。然而，它在特定的国际化环境中并不能正常工作。如果你已经是老司机的话，你可以选用你最喜欢的编辑器。

出于这种考虑，我打算让你尝试一些你所在平台上的标准的用于编程的文本编辑器，并且长期使用其中你最喜欢的一个。如果你已经用了Gedit并且很喜欢他，那么就一致用下去。如果你打算尝试一些不同的编辑器，则赶快尝试并选择一个。

最重要的事情是，不要纠结于寻找最完美的编辑器。文本编辑器几乎都很奇怪，你只需要选择一个并熟悉它，如果你发现喜欢别的编辑器可以切换到它。不要在挑选它和把它变得更好上面花很多时间。

这是亦可以尝试的一些编辑器：

- Linux和OSX上的 `Gedit` 。
- OSX上的 `TextWrangler` 。
- 可以在终端中运行并几乎在任何地方工作的 `Nano` 。
- `Emacs` 和 `Emacs OSX` 。需要学习一些东西。
- `Vim` 和 `Mac Vim` 。

每个人都可能选择一款不同的编辑器，这些只是一部分人所选择的开源编辑器。在找到你最喜欢的那个之前，尝试其中的一些，甚至是一些商业编辑器。

警告：不要使用IDE

IDE，或者“集成开发工具”，会使你变笨。如果你想要成为一个好的程序员，它会是最糟糕的工具，因为它隐藏了背后的细节，你的工作是弄清楚背后发生了什么。如果你试着完成一些事情，并且所在平台根据特定的IDE而设计，它们非常有用，但是对于学习C编程（以及许多其它语言），它们没有意义。

> 注
>
> 如果你玩过吉他，你应该知道TAB是什么。但是对于其它人，让我对其做个解释。在音乐中有一种乐谱叫做“五线谱”。它是通用、非常古老的乐谱，以一种通用的方法来记下其它人应该在乐器上弹奏的音符。如果你弹过钢琴，这种乐谱非常易于使用，因为它几乎就是为钢琴和交响乐发明的。
>
> 然而吉他是一种奇怪的乐器，它并不能很好地适用这种乐谱。所以吉他手通常使用一种叫做TAB（tablature）的乐谱。它所做的不是告诉你该弹奏哪个音符，而是在当时应该拨哪根弦。你完全可以在不知道所弹奏的单个音符的情况下学习整首乐曲，许多人也都是这么做的，但是如果你想知道你弹的是什么，TAB是毫无意义的。
>
> 传统的乐谱可能比TAB更难一些，但是会告诉你如何演奏音乐，而不是如果玩吉他。通过传统的乐谱我可以在钢琴上，或者在贝斯上弹奏相同的曲子。我也可以将它放到电脑中，为它设计全部的曲谱。但是通过TAB我只能在吉他上弹奏。
>
> IDE就像是TAB，你可以用它非常快速地编程，但是你只能够用一种语言在一个平台上编程。这就是公司喜欢将它卖给你的原因。它们知道你比较懒，并且由于它只适用于它们自己的平台，他们就将你锁定在了那个平台上。
>
> 打破这一循环的办法就是不用IDE学习编程。一个普通的文本编辑器，或者一个程序员使用的文本编辑器，例如Vim或者Emacs，能让你更熟悉代码。这有一点点困难，但是终结果是你将会熟悉任何代码，在任何计算机上，以任何语言，并且懂得背后的原理。

练习1：启用编译器

> 原文：Exercise 1: Dust Off That Compiler
>
> 译者：飞龙

这是你用C写的第一个简单的程序：

```
int main(int argc, char *argv[])
{
    puts("Hello world.");

    return 0;
}
```

把它写进 `ex1.c` 并输入：

```
$ make ex1
cc     ex1.c   -o ex1
```

你的编译器可能会使用一个有些不同的命令，但是最后应该会产生一个名为 `ex1` 的文件，并且你可以运行它。

你会看到什么

现在你可以运行程序并看到输出。

```
$ ./ex1
Hello world.
```

如果没有，则需要返回去修复它。

如何使它崩溃

在这本书中我会添加一个小节，关于如何使程序崩溃。我会让你对程序做一些奇怪的事情，以奇怪的方式运行，或者修改代码，以便让你看到崩溃和编译器错误。

对于这个程序，打开所有编译警告重新构建它：

```
$ rm ex1
$ CFLAGS="-Wall" make ex1
cc -Wall    ex1.c    -o ex1
ex1.c: In function 'main':
ex1.c:3: warning: implicit declaration of function 'puts'
$ ./ex1
Hello world.
$
```

现在你会得到一个警告，说 `puts` 函数是隐式声明的。C语言的编译器很智能，它能够理解你想要什么。但是如果可以的话，你应该去除所有编译器警告。把下面一行添加到 `ex1.c` 文件的最上面，之后重新编译来去除它：

```
#include <stdio.h>
```

现在像刚才一样重新执行make命令，你会看到所有警告都消失了。

附加题

- 在你的文本编辑器中打开 `ex1` 文件，随机修改或删除一部分，之后运行它看看发生了什么。
- 再多打印5行文本或者其它比 `"Hello world."` 更复杂的东西。
- 执行 `man 3 puts` 来阅读这个函数和其它函数的文档。

练习2：用Make来代替Python

> 原文：[Exercise 2: Make Is Your Python Now]
>
> 译者：飞龙

在Python中，你仅仅需要输入 `python` ，就可以运行你想要运行的代码。Python的解释器会运行它们，并且在运行中导入它所需的库和其它东西。C是完全不同的东西，你需要事先编译你的源文件，并且手动将它们整合为一个可以自己运行的二进制文件。手动来做这些事情很痛苦，在上一个练习中只需要运行 `make` 就能完成。

这个练习是GNU make 的速成课，由于你在学C语言，所以你就必须掌握它。Make将贯穿剩下的课程，等效于Python（命令）。它会构建源码，执行测试，设置一些选项以及为你做所有Python通常会做的事情。

有所不同的是，我会向你展示一些更智能化的Makefile魔法，你不需要指出你的C程序的每一个愚蠢的细节来构建它。我不会在练习中那样做，但是你需要先用一段时间的“低级 make”，我才能向你演示“大师级的make”。

使用 Make

使用make的第一阶段就是用它已知的方式来构建程序。Make预置了一些知识，来从其它文件构建多种文件。上一个练习中，你已经使用像下面的命令来这样做了：

```
$ make ex1
# or this one too
$ CFLAGS="-Wall" make ex1
```

第一个命令中你告诉make，“我想创建名为ex1的文件”。于是Make执行下面的动作：

- 文件 `ex1` 存在吗？
- 没有。好的，有没有其他文件以 `ex1` 开头？
- 有，叫做 `ex1.c` 。我知道如何构建 `.c` 文件吗？
- 是的，我会运行命令 `cc ex1.c -o ex1` 来构建它。
- 我将使用 `cc` 从 `ex1.c` 文件来为你构建 `ex1` 。

上面列出的第二条命令是一种向make命令传递“修改器”的途径。如果你不熟悉Unix shell如何工作，你可以创建这些“环境变量”，它们会在程序运行时生效。有时你会用一条类似于 `export CFLAGS="-Wall"` 的命令来执行相同的事情，取决于你所用的shell。然而你可以仅仅把它们放到你想执行的命令前面，于是环境变量只会在程序运行时有效。

在这个例子中我执行了 `CFLAGS="-Wall" make ex1` ，所以它会给make通常使用的 `cc` 命令添加 `-Wall` 选项。这行命令告诉 `cc` 编译器要报告所有的警告（然而实际上不可能报告所有警告）。

实际上你可以深入探索使用make的上述方法，但是先让我们来看看 `Makefile` ，以便让你对make了解得更多一点。首先，创建文件并写入以下内容：

```
CFLAGS=-Wall -g

clean:
	rm -f ex1
```

将文件在你的当前文件夹上保存为 `Makefile` 。Make会自动假设当前文件夹中有一个叫做 `Makefile` 的文件，并且会执行它。此外，一定要注意：确保你只输入了 TAB 字符，而不是空格和 TAB 的混合。

> 译者注：上述代码中第四行 `rm` 前面是一个 TAB ，而不是多个等量的空格。

`Makefile` 向你展示了make的一些新功能。首先我们在文件中设置 `CFLAGS` ，所以之后就不用再设置了。并且，我们添加了 `-g` 标识来获取调试信息。接着我们写了一个叫做 `clean` 的部分，它告诉make如何清理我们的小项目。

确保它和你的 `ex1.c` 文件在相同的目录中，之后运行以下命令：

```
$ make clean
$ make ex1
```

你会看到什么

如果代码能正常工作，你应该看到这些：

```
$ make clean
rm -f ex1
$ make ex1
cc -Wall -g    ex1.c   -o ex1
ex1.c: In function 'main':
ex1.c:3: warning: implicit declaration of function 'puts'
$
```

你可以看出来我执行了 `make clean` ，它告诉make执行我们的 `clean` 目标。再去看一眼Makefile，之后你会看到在它的下面，我缩进并且输入了一些想要make为我运行的shell命令。你可以在此处输入任意多的命令，所以它是一个非常棒的自动化工具。

> 注
>
> 如果你修改了`ex1.c`，添加了`#include<stdio>`，输出中的关于`puts`的警告就会消失（这其实应该算作一个错误）。我这里有警告是因为我并没有去掉它。

同时也要注意，即使我们在`Makefile`中并没有提到`ex1`，`make`仍然会知道如何构建它，以及使用我们指定的设置。

如何使它崩溃

上面那些已经足够让你起步了，但是让我们以一种特定的方式来破坏make文件，以便你可以看到发生了什么。找到`rm -f ex1`的那一行并去掉缩进（让它左移），之后你可以看到发生了什么。再次运行`make clean`，你就会得到下面的信息：

```
$ make clean
Makefile:4: *** missing separator.  Stop.
```

永远记住要缩进，以及如果你得到了像这种奇奇怪怪的错误，应该复查你是否都使用了 TAB 字符，由于一些make的变种十分挑剔。

附加题

- 创建目标`all:ex1`，可以以单个命令`make`构建`ex1`。
- 阅读`man make`来了解关于如何执行它的更多信息。
- 阅读`man cc`来了解关于`-Wall`和`-g`行为的更多信息。
- 在互联网上搜索Makefile文件，看看你是否能改进你的文件。
- 在另一个C语言项目中找到`Makefile`文件，并且尝试理解它做了什么。

练习3：格式化输出

> 原文：Exercise 3: Formatted Printing
>
> 译者：飞龙

不要删除Makefile，因为它可以帮你指出错误，以及当我们需要自动化处理一些事情时，可以向它添加新的东西。

许多编程语言都使用了C风格的格式化输出，所以让我们尝试一下：

```
#include <stdio.h>

int main()
{
    int age = 10;
    int height = 72;

    printf("I am %d years old.\n", age);
    printf("I am %d inches tall.\n", height);

    return 0;
}
```

写完之后，执行通常的 `make ex3` 命令来构建并运行它。一定要确保你处理了所有的警告。

这个练习的代码量很小，但是信息量很大，所以让我们逐行分析一下：

- 首先你包含了另一个头文件叫做 `stdio.h` 。这告诉了编译器你要使用“标准的输入/输出函数”。它们之一就是 `printf` 。
- 然后你使用了一个叫 `age` 的变量并且将它设置为10。
- 接着你使用了一个叫 `height` 的变量并且设置为72。
- 再然后你使用 `printf` 函数来打印这个星球上最高的十岁的人的年龄和高度。
- 在 `printf` 中你会注意到你传入了一个字符串，这就是格式字符串，和其它语言中一样。
- 在格式字符串之后，你传入了一些变量，它们应该被 `printf` “替换”进格式字符串中。

这些语句的结果就是你用 `printf` 处理了一些变量，并且它会构造出一个新的字符串，之后将它打印在终端上。

你会看到什么

当你做完上面的整个步骤，你应该看到这些东西：

```
$ make ex3
cc -Wall -g    ex3.c   -o ex3
$ ./ex3
I am 10 years old.
I am 72 inches tall.
$
```

不久之后我会停下来让你运行 `make` ，并且告诉你构建过程是什么样子的。所以请确保你正确得到了这些信息并且能正常执行。

外部研究

在附加题一节我可能会让你自己查找一些资料，并且弄明白它们。这对于一个自我学习的程序员来说相当重要。如果你一直在自己尝试了解问题之前去问其它人，你永远都不会学到独立解决问题。这会让你永远都不会在自己的技能上建立信心，并且总是依赖别人去完成你的工作。

打破你这一习惯的方法就是强迫你自己先试着自己回答问题，并且确认你的回答是正确的。你可以通过打破一些事情，用实验验证可能的答案，以及自己进行研究来完成它。

对于这个练习，我想让你上网搜索 `printf` 的所有格式化占位符和转义序列。转义序列类似 `\n` 或者 `\r` ，可以让你分别打印新的一行或者 tab 。格式化占位符类似 `%s` 或者 `%d` ，可以让你打印字符串或整数。找到所有的这些东西，以及如何修改它们，和可设置的“精度”和宽度的种类。

从现在开始，这些任务会放到附加题里面，你应该去完成它们。

如何使它崩溃

尝试下面的一些东西来使你的程序崩溃，在你的电脑上它们可能会崩溃，也可能不会。

- 从第一个 `printf` 中去掉 `age` 并重新编译，你应该会得到一大串的警告。
- 运行新的程序，它会崩溃，或者打印出奇怪的年龄。
- 将 `printf` 恢复原样，并且去掉 `age` 的初值，将那一行改为 `int age;` ，之后重新构建并运行。

```
# edit ex3.c to break printf
$ make ex3
cc -Wall -g    ex3.c   -o ex3
ex3.c: In function 'main':
ex3.c:8: warning: too few arguments for format
ex3.c:5: warning: unused variable 'age'
$ ./ex3
I am -919092456 years old.
I am 72 inches tall.
# edit ex3.c again to fix printf, but don't init age
$ make ex3
cc -Wall -g    ex3.c   -o ex3
ex3.c: In function 'main':
ex3.c:8: warning: 'age' is used uninitialized in this function
$ ./ex3
I am 0 years old.
I am 72 inches tall.
$
```

附加题

- 找到尽可能多的方法使`ex3`崩溃。
- 执行`man 3 printf`来阅读其它可用的'%'格式化占位符。如果你在其它语言中使用过它们，应该看着非常熟悉（它们来源于`printf`）。
- 将`ex3`添加到你的`Makefile`的`all`列表中。到目前为止，可以使用`make clean all`来构建你所有的练习。
- 将`ex3`添加到你的`Makefile`的`clean`列表中。当你需要的时候使用`make clean`可以删除它。

练习4：Valgrind 介绍

> 原文：[Exercise 4: Introducing Valgrind](#)
>
> 译者：[飞龙](#)

现在是介绍另一个工具的时间了，在你学习C的过程中，你会时时刻刻用到它，它就是 `Valgrind` 。我现在就向你介绍 `Valgrind` ，是因为从现在开始你将会在“如何使它崩溃”一节中用到它。 `Valgrind` 是一个运行你的程序的程序，并且随后会报告所有你犯下的可怕错误。它是一款相当棒的自由软件，我在编写C代码时一直使用它。

回忆一下在上一章中，我让你移除 `printf` 的一个参数，来使你的代码崩溃。它打印出了一些奇怪的结果，但我并没有告诉你为什么它会这样打印。这个练习中我们要使用 `Valgrind` 来搞清楚为什么。

> 注
>
> 这本书的前几章讲解了一小段代码，同时掺杂了一些必要的工具，它们在本书的剩余章节会用到。这样做的原因是，阅读这本书的大多数人都不熟悉编译语言，也必然不熟悉自动化的辅助工具。通过先让你懂得如何使用 `make` 和 `Valgrind` ，我可以在后面使用它们更快地教你C语言，以及帮助你尽早找出所有的bug。
>
> 这一章之后我就不再介绍更多的工具了，每章的内容大部分是代码，以及少量的语法。然而，我也会提及少量工具，我们可以用它来真正了解发生了什么，以及更好地了解常见的错误和问题。

安装 Valgrind

你可以用OS上的包管理器来安装 `Valgrind` ，但是我想让你学习如何从源码安装程序。这涉及到下面几个步骤：

- 下载源码的归档文件来获得源码
- 解压归档文件，将文件提取到你的电脑上
- 运行 `./configure` 来建立构建所需的配置
- 运行 `make` 来构建源码，就像之前所做的那样
- 运行 `sudo make install` 来将它安装到你的电脑

下面是执行以上步骤的脚本，我想让你复制它：

```
# 1) Download it (use wget if you don't have curl)
curl -O http://valgrind.org/downloads/valgrind-3.6.1.tar.bz2

# use md5sum to make sure it matches the one on the site
md5sum valgrind-3.6.1.tar.bz2

# 2) Unpack it.
tar -xjvf valgrind-3.6.1.tar.bz2

# cd into the newly created directory
cd valgrind-3.6.1

# 3) configure it
./configure

# 4) make it
make

# 5) install it (need root)
sudo make install
```

按照这份脚本，但是如果 `Valgrind` 有新的版本请更新它。如果它不能正常执行，也请试着深入研究原因。

使用 Valgrind

使用 `Valgrind` 十分简单，只要执行 `valgrind theprogram` ，它就会运行你的程序，随后打印出你的程序运行时出现的所有错误。在这个练习中，我们会崩溃在一个错误输出上，然后会修复它。

首先，这里有一个 `ex3.c` 的故意出错的版本，叫做 `ex4.c` 。出于练习目的，将它再次输入到文件中：

```c
#include <stdio.h>

/* Warning: This program is wrong on purpose. */

int main()
{
    int age = 10;
    int height;

    printf("I am %d years old.\n");
    printf("I am %d inches tall.\n", height);

    return 0;
}
```

你会发现，除了两个经典的错误外，其余部分都相同：

- 没有初始化 `height` 变量
- 没有将 `age` 变量传入第一个 `printf` 函数

你会看到什么

现在我们像通常一样构建它，但是不要直接运行，而是使用 `Valgrind` 来运行它（见源码："使用Valgrind构建并运行 ex4.c"）：

```
$ make ex4
cc -Wall -g    ex4.c   -o ex4
ex4.c: In function 'main':
ex4.c:10: warning: too few arguments for format
ex4.c:7: warning: unused variable 'age'
ex4.c:11: warning: 'height' is used uninitialized in this functi
on
$ valgrind ./ex4
==3082== Memcheck, a memory error detector
==3082== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Sewar
d et al.
==3082== Using Valgrind-3.6.0.SVN-Debian and LibVEX; rerun with
-h for copyright info
==3082== Command: ./ex4
==3082==
I am -16775432 years old.
==3082== Use of uninitialised value of size 8
==3082==    at 0x4E730EB: _itoa_word (_itoa.c:195)
==3082==    by 0x4E743D8: vfprintf (vfprintf.c:1613)
==3082==    by 0x4E7E6F9: printf (printf.c:35)
==3082==    by 0x40052B: main (ex4.c:11)
==3082==
==3082== Conditional jump or move depends on uninitialised value
(s)
==3082==    at 0x4E730F5: _itoa_word (_itoa.c:195)
==3082==    by 0x4E743D8: vfprintf (vfprintf.c:1613)
==3082==    by 0x4E7E6F9: printf (printf.c:35)
==3082==    by 0x40052B: main (ex4.c:11)
==3082==
==3082== Conditional jump or move depends on uninitialised value
(s)
==3082==    at 0x4E7633B: vfprintf (vfprintf.c:1613)
==3082==    by 0x4E7E6F9: printf (printf.c:35)
==3082==    by 0x40052B: main (ex4.c:11)
==3082==
==3082== Conditional jump or move depends on uninitialised value
(s)
==3082==    at 0x4E744C6: vfprintf (vfprintf.c:1613)
==3082==    by 0x4E7E6F9: printf (printf.c:35)
==3082==    by 0x40052B: main (ex4.c:11)
```

```
==3082==
I am 0 inches tall.
==3082==
==3082== HEAP SUMMARY:
==3082==     in use at exit: 0 bytes in 0 blocks
==3082==   total heap usage: 0 allocs, 0 frees, 0 bytes allocate
d
==3082==
==3082== All heap blocks were freed -- no leaks are possible
==3082==
==3082== For counts of detected and suppressed errors, rerun wit
h: -v
==3082== Use --track-origins=yes to see where uninitialised valu
es come from
==3082== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 4
from 4)
$
```

> 注
>
> 如果你运行了 `Valgrind` ，它显示一些类似于 `by 0x4052112: (below main) (libc-start.c:226)` 的东西，而不是 `main.c` 中的行号，你需要使用 `valgrind --track-origins=yes ./ex4` 命令来运行你的 `Valgrind` 。由于某些原因， `valgrind` 的Debian和Ubuntu上的版本会这样，但是其它的不会。

上面那段输出非常长，因为 `Valgrind` 在明确地告诉你程序中的每个错误都在哪儿。让我们从开头逐行分析一下（行号在左边，你可以参照）：

1

你执行了通常的 `make ex4` 来构建它。确保你看到的 `cc` 命令和它一样，并且带有 `-g` 选项，否则 `Valgrind` 的输出不会带上行号。

2~6

要注意编译器也会向你报告源码的错误，它警告你“向格式化函数传入了过少的变量”，因为你忘记包含 `age` 变量。

7

然后使用 `valgrind ./ex4` 来运行程序。

8

之后 `Valgrind` 变得十分奇怪，并向你报错：

14~18

在 `main (ex4.c:11)` （意思是文件 `ex4.c` 的 `main` 函数的第11行）的那行中，有“大小为8的未初始化的值”。你通过查看错误找到了它，并且在它下面看到了“栈踪迹”。最开始看到的那行 `(ex4.c:11)` 在最下面，如果你不明白哪里出错了，你可以向上看，比如 `printf.c:35` 。通常最下面的一行最重要（这个例子中是第18行）。

20~24

下一个错误位于 `main` 函数中的 `ex4.c:11` 。 `Valgrind` 不喜欢这一行，它说的是一些 if 语句或者 while 循环基于一个未初始化的值，在这个例子中是 `height` 。

25~35

剩下的错误都大同小异，因为这个值还在继续使用。

37~46

最后程序退出了， `Valgrind` 显示出一份摘要，告诉你程序有多烂。

这段信息读起来会相当多，下面是你的处理方法：

- 无论什么时候你运行C程序并且使它工作，都应该使用 `Valgrind` 重新运行它来检查。
- 对于得到的每个错误，找到“源码:行数”提示的位置，然后修复它。你可以上网搜索错误信息，来弄清楚它的意思。
- 一旦你的程序在 `Valgrind` 下不出现任何错误信息，应该就好了。你可能学会了如何编写代码的一些技巧。

在这个练习中我并不期待你马上完全掌握 `Valgrind` ，但是你应该安装并且学会如何快速使用它，以便我们将它用于后面的练习。

附加题

- 按照上面的指导，使用 `Valgrind` 和编译器修复这个程序。
- 在互联网上查询 `Valgrind` 相关的资料。
- 下载另一个程序并手动构建它。尝试一些你已经使用，但从来没有手动构建的程序。
- 看看 `Valgrind` 的源码是如何在目录下组织的，并且阅读它的Makefile文件。不要担心，这对我来说没有任何意义。

练习5：一个C程序的结构

> 原文：Exercise 5: The Structure Of A C Program
>
> 译者：飞龙

你已经知道了如何使用`printf`，也有了可以随意使用的一些工具，现在让我们逐行分析一个简单的C程序，以便你了解它是如何组织的。在这个程序里你会编写一些不是很熟悉的东西，我会轻松地把它们拆开。之后在后面的几章我们将会处理这些概念。

```
#include <stdio.h>

/* This is a comment. */
int main(int argc, char *argv[])
{
    int distance = 100;

    // this is also a comment
    printf("You are %d miles away.\n", distance);

    return 0;
}
```

手动输入这段代码并运行它，之后确保在`Valgrind`下不出现任何错误。你可能不会这样做，但你得习惯它。

你会看到什么

这真是一段无聊的输出，但是这个练习的目的是让你分析代码：

```
$ make ex5
cc -Wall -g    ex5.c   -o ex5
$ ./ex5
You are 100 miles away.
$
```

分解代码

当你输出这段代码时，可能你只弄清楚了这段代码中的一小部分C语言特性。让我们快速地逐行分解它，之后我们可以做一些练习来更好地了解每一部分：

- ex5.c:1

这是一个 `include` ，它是将一个文件的内容导入到这个文件的方式。C具有使用 `.h` 扩展名作为头文件的惯例。头文件中拥有一些函数的列表，这些都是你想在程序中使用的函数。

ex5.c:3

这是多行注释，你可以在 `/*` 和 `*/` 之间放置任意多行。

ex5.c:4

这是一个你遇到的更复杂的 `main` 函数。操作系统加载完你的程序，之后会运行叫做 `main` 的函数，这是C程序的工作方式。这个函数只需要返回 `int` ，并接受两个参数，一个是 `int` 作为命令行参数的数量，另一个是 `char*` 字符串的数组作为命令行参数。这是不是让人难以理解？不用担心，我们稍后会讲解它。

ex5.c:5

任何函数都以 `{` 字符开始，它表示“程序块”的开始。在Python中用一个 `:` 来表示。在其它语言中，可能需要用 `begin` 或者 `do` 来表示。

ex5.c:6

一个变量的声明和同时的赋值。你可以使用语法 `type name = value;` 来创建变量。在C的语句中，除了逻辑语句，都以一个 `;` （分号）来结尾。

ex5.c:8

注释的另一种形式，它就像Python或Ruby的注释。它以 `//` 开头，直到行末结束。

ex5.c:9

调用了我们的老朋友 `printf` 。就像许多语言中的函数调用，使用语法 `name(arg1, arg2);` 。函数可以不带任何参数，也可以拥有任何数量的参数。 `printf` 函数是一类特别的函数，可以带可变数量的参数。我们会在之后说明。

ex5.c:11

一个 `main` 函数的返回语句，它会向OS提供退出值。你可能不熟悉Unix软件的返回代码，所以这个也放到后面去讲。

ex5.c:12

最后，我们以一个闭合的 `}` 花括号来结束了 `main` 函数。它就是整个程序的结尾了。

在这次分解中有大量的信息，所以你应该逐行来学习，并且确保至少掌握了背后发生了什么。你不一定了解所有东西，但是在我们继续之前，你可以猜猜它们的意思。

附加题

- 对于每一行，写出你不理解的符号，并且看看是否能猜出它们的意思。在纸上写下你的猜测，你可以在以后检查它，看看是否正确。
- 回头去看之前几个练习的源代码，并且像这样分解代码，来看看你是否了解它们。写下你不了解和不能自己解释的东西。

练习6：变量类型

> 原文：Exercise 6: Types Of Variables
>
> 译者：飞龙

你应该掌握了一个简单的C程序的结构，所以让我们执行下一步简单的操作，声明不同类型的变量。

```
include <stdio.h>

int main(int argc, char *argv[])
{
    int distance = 100;
    float power = 2.345f;
    double super_power = 56789.4532;
    char initial = 'A';
    char first_name[] = "Zed";
    char last_name[] = "Shaw";

    printf("You are %d miles away.\n", distance);
    printf("You have %f levels of power.\n", power);
    printf("You have %f awesome super powers.\n", super_power);
    printf("I have an initial %c.\n", initial);
    printf("I have a first name %s.\n", first_name);
    printf("I have a last name %s.\n", last_name);
    printf("My whole name is %s %c. %s.\n",
            first_name, initial, last_name);

    return 0;
}
```

在这个程序中我们声明了不同类型的变量，并且使用了不同的 `printf` 格式化字符串来打印它们。

你会看到什么

你的输出应该和我的类似，你可以看到C的格式化字符串相似于Python或其它语言，很长一段时间中都是这样。

```
$ make ex6
cc -Wall -g    ex6.c   -o ex6
$ ./ex6
You are 100 miles away.
You have 2.345000 levels of power.
You have 56789.453200 awesome super powers.
I have an initial A.
I have a first name Zed.
I have a last name Shaw.
My whole name is Zed A. Shaw.
$
```

你可以看到我们拥有一系列的“类型”，它们告诉编译器变量应该表示成什么，之后格式化字符串会匹配不同的类型。下面解释了它们如何匹配：

整数

使用`int`声明，使用`%d`来打印。

浮点

使用`float`或`double`声明，使用`%f`来打印。

字符

使用`char`来声明，以周围带有`'`（单引号）的单个字符来表示，使用`%c`来打印。

字符串（字符数组）

使用`char name[]`来声明，以周围带有`"`的一些字符来表示，使用`%s`来打印。

你会注意到C语言中区分单引号的`char`和双引号的`char[]`或字符串。

> 注
>
> 当我提及C语言类型时，我通常会使用`char[]`来代替整个的`char SOMENAME[]`。这不是有效的C语言代码，只是一个用于讨论类型的一个简化表达方式。

如何使它崩溃

你可以通过向`printf`传递错误的参数来轻易使这个程序崩溃。例如，如果你找到打印我的名字的那行，把`initial`放到`first_name`前面，你就制造了一个bug。执行上述修改编译器就会向你报错，之后运行的时候你可能会得到一个“段错误”，就像这样：

```
$ make ex6
cc -Wall -g    ex6.c   -o ex6
ex6.c: In function 'main':
ex6.c:19: warning: format '%s' expects type 'char *', but argume
nt 2 has type 'int'
ex6.c:19: warning: format '%c' expects type 'int', but argument
3 has type 'char *'
$ ./ex6
You are 100 miles away.
You have 2.345000 levels of power.
You have 56789.453125 awesome super powers.
I have an initial A.
I have a first name Zed.
I have a last name Shaw.
Segmentation fault
$
```

在 `Valgrind` 下运行修改后的程序，来观察它会告诉你什么关于错误“Invalid read of size 1”的事情。

附加题

- 寻找其他通过修改 `printf` 使这段C代码崩溃的方法。
- 搜索“ `printf` 格式化”，试着使用一些高级的占位符。
- 研究可以用几种方法打印数字。尝试以八进制或十六进制打印，或者其它你找到的方法。
- 试着打印空字符串，即 `""` 。

练习7：更多变量和一些算术

> 原文：Exercise 7: More Variables, Some Math
>
> 译者：飞龙

你可以通过声明`int`，`float`，`char`和`double`类型的变量，来对它们做更多的事情，让我们来熟悉它们吧。接下来我们会在各种数学表达式中使用它们，所以我会向你介绍C的基本算术操作。

```
int main(int argc, char *argv[])
{
    int bugs = 100;
    double bug_rate = 1.2;

    printf("You have %d bugs at the imaginary rate of %f.\n",
            bugs, bug_rate);

    long universe_of_defects = 1L * 1024L * 1024L * 1024L;
    printf("The entire universe has %ld bugs.\n",
            universe_of_defects);

    double expected_bugs = bugs * bug_rate;
    printf("You are expected to have %f bugs.\n",
            expected_bugs);

    double part_of_universe = expected_bugs / universe_of_defect
s;
    printf("That is only a %e portion of the universe.\n",
            part_of_universe);

    // this makes no sense, just a demo of something weird
    char nul_byte = '\0';
    int care_percentage = bugs * nul_byte;
    printf("Which means you should care %d%%.\n",
            care_percentage);

    return 0;
}
```

下面是这一小段无意义代码背后发生的事情：

ex7.c:1-4

C程序的通常开始。

ex7.c:5-6

为一些伪造的bug数据声明了一个`int`和一个`double`变量。

ex7.c:8-9

打印这两个变量，没有什么新东西。

ex7.c:11

使用了一个新的类型 `long` 来声明一个大的数值，它可以储存比较大的数。

ex7.c:12-13

使用 `%ld` 打印出这个变量，我们添加了个修饰符到 `%d` 上面。添加的"l"表示将它当作长整形打印。

ex7.c:15-17

只是更多的算术运算和打印。

ex7.c:19-21

编撰了一段你的bug率的描述，这里的计算非常不精确。结果非常小，所以我们要使用 `%e` 以科学记数法的形式打印它。

ex7.c:24

以特殊的语法 `'\0'` 声明了一个字符。这样创建了一个“空字节”字符，实际上是数字0。

ex7.c:25

使用这个字符乘上bug的数量，它产生了0，作为“有多少是你需要关心的”的结果。这条语句展示了你有时会碰到的丑陋做法。

ex7.c:26-27

将它打印出来，注意我使用了 `%%` （两个百分号）来打印一个 `%` 字符。

ex7.c:28-30

`main` 函数的结尾。

这一段代码只是个练习，它演示了许多算术运算。在最后，它也展示了许多你能在C中看到，但是其它语言中没有的技巧。对于C来说，一个“字符”同时也是一个整数，虽然它很小，但的确如此。这意味着你可以对它做算术运算，无论是好是坏，许多软件中也是这样做的。

在最后一部分中，你第一次见到C语言是如何直接访问机器的。我们会在后面的章节中深入。

你会看到什么

通常，你应该看到如下输出：

```
$ make ex7
cc -Wall -g    ex7.c   -o ex7
$ ./ex7
You have 100 bugs at the imaginary rate of 1.200000.
The entire universe has 1073741824 bugs.
You are expected to have 120.000000 bugs.
That is only a 1.117587e-07 portion of the universe.
Which means you should care 0%.
$
```

如何使它崩溃

像之前一样，向`printf`传入错误的参数来使它崩溃。对比`%c`，看看当你使用`%s`来打印`nul_byte`变量时会发生什么。做完这些之后，在`Valgrind`下运行它看看关于你的这次尝试会输出什么。

附加题

- 把为`universe_of_defects`赋值的数改为不同的大小，观察编译器的警告。
- 这些巨大的数字实际上打印成了什么？
- 将`long`改为`unsigned long`，并试着找到对它来说太大的数字。
- 上网搜索`unsigned`做了什么。
- 试着自己解释（在下个练习之前）为什么`char`可以和`int`相乘。

练习8：大小和数组

> 原文：Exercise 8: Sizes And Arrays
>
> 译者：飞龙

在上一个练习中你做了一些算术运算，并且使用了 `'\0'` （空）字符。这对于其它语言来说非常奇怪，因为它们把“字符串”和“字节数组”看做不同的东西。但是C中的字符串就是字节数组，并且只有不同的打印函数才知道它们的不同。

在我真正解释其重要性之前，我先要介绍一些概念： `sizeof` 和数组。下面是我们将要讨论的一段代码：

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int areas[] = {10, 12, 13, 14, 20};
    char name[] = "Zed";
    char full_name[] = {
        'Z', 'e', 'd',
         ' ', 'A', '.', ' ',
         'S', 'h', 'a', 'w', '\0'
    };

    // WARNING: On some systems you may have to change the
    // %ld in this code to a %u since it will use unsigned ints
    printf("The size of an int: %ld\n", sizeof(int));
    printf("The size of areas (int[]): %ld\n",
            sizeof(areas));
    printf("The number of ints in areas: %ld\n",
            sizeof(areas) / sizeof(int));
    printf("The first area is %d, the 2nd %d.\n",
            areas[0], areas[1]);

    printf("The size of a char: %ld\n", sizeof(char));
    printf("The size of name (char[]): %ld\n",
            sizeof(name));
    printf("The number of chars: %ld\n",
            sizeof(name) / sizeof(char));

    printf("The size of full_name (char[]): %ld\n",
            sizeof(full_name));
    printf("The number of chars: %ld\n",
            sizeof(full_name) / sizeof(char));

    printf("name=\"%s\" and full_name=\"%s\"\n",
            name, full_name);

    return 0;
}
```

这段代码中我们创建了一些不同数据类型的数组。由于数组是C语言工作机制的核心，有大量的方法可以用来创建数组。我们暂且使用 `type name[] = {initializer};` 语法，之后我们会深入研究。这个语法的意思是，“我想要那个类型的数组并且初始化为{..}”。C语言看到它时，会做这些事情：

- 查看它的类型，以第一个数组为例，它是 `int` 。
- 查看 `[]` ，看到了没有提供长度。
- 查看初始化表达式 `{10, 12, 13, 14, 20}` ，并且了解你想在数组中存放这5个整数。
- 在电脑中开辟出一块空间，可以依次存放这5个整数。

- 将数组命名为 `areas` ，也就是你想要的名字，并且在当前位置给元素赋值。

在 `areas` 的例子中，我们创建了一个含有5个整数的数组来存放那些数字。当它看到 `char name[] = "Zed";` 时，它会执行相同的步骤。我们先假设它创建了一个含有3个字符的数组，并且把字符赋值给 `name` 。我们创建的最后一个数组是 `full_name` ，但是我们用了一个比较麻烦的语法，每次用一个字符将其拼写出来。对C来说， `name` 和 `full_name` 的方法都可以创建字符数组。

在文件的剩余部分，我们使用了 `sizeof` 关键字来问C语言这些东西占多少个字节。C语言无非是内存块的大小和地址以及在上面执行的操作。它向你提供了 `sizeof` 便于你理解它们，所以你在使用一个东西之前可以先询问它占多少空间。

这是比较麻烦的地方，所以我们先运行它，之后再解释。

你会看到什么

```
$ make ex8
cc -Wall -g    ex8.c   -o ex8
$ ./ex8
The size of an int: 4
The size of areas (int[]): 20
The number of ints in areas: 5
The first area is 10, the 2nd 12.
The size of a char: 1
The size of name (char[]): 4
The number of chars: 4
The size of full_name (char[]): 12
The number of chars: 12
name="Zed" and full_name="Zed A. Shaw"
$
```

现在你可以看到这些不同 `printf` 调用的输出，并且瞥见C语言是如何工作的。你的输出实际上可能会跟我的完全不同，因为你电脑上的整数大小可能会不一样。下面我会过一遍我的输出：

> 译者注：16位机器上的 `int` 是16位的，不过现在16位机很少见了吧。

5

我的电脑认为 `int` 的大小是4个字节。你的电脑上根据位数不同可能会使用不同的大小。

6

`areas` 中含有5个整数，所以我的电脑自然就需要20个字节来储存它。

7

如果我们把`areas`的大小与`int`的大小相除，我们就会得到元素数量为5。这也符合我们在初始化语句中所写的东西。

8

接着我们访问了数组，读出`areas[0]`和`areas[1]`，这也意味着C语言的数组下标是0开头的，像Python和Ruby一样。

9~11

我们对`name`数组执行同样的操作，但是注意到数组的大小有些奇怪，它占4个字节，但是我们用了三个字符来打出"Zed"。那么第四个字符是哪儿来的呢？

12~13

我们对`full_name`数组执行了相同的操作，但它是正常的。

13

最后我们打印出`name`和`full_name`，根据`printf`证明它们实际上就是“字符串”。

确保你理解了上面这些东西，并且知道这些输出对应哪些创建的变量。后面我们会在它的基础上探索更多关于数组和存储空间的事情。

如何使它崩溃

使这个程序崩溃非常容易，只需要尝试下面这些事情：

- 将`full_name`最后的`'\0'`去掉，并重新运行它，在`valgrind`下再运行一遍。现在将`full_name`的定义从`main`函数中移到它的上面，尝试在`Valgrind`下运行它来看看是否能得到一些新的错误。有些情况下，你会足够幸运，不会得到任何错误。
- 将`areas[0]`改为`areas[10]`并打印，来看看`Valgrind`会输出什么。
- 尝试上述操作的不同变式，也对`name`和`full_name`执行一遍。

附加题

- 尝试使用`areas[0] = 100;`以及相似的操作对`areas`的元素赋值。
- 尝试对`name`和`full_name`的元素赋值。
- 尝试将`areas`的一个元素赋值为`name`中的字符。
- 上网搜索在不同的CPU上整数所占的不同大小。

练习9：数组和字符串

> 原文：Exercise 9: Arrays And Strings
>
> 译者：飞龙

上一个练习中，我们学习了如何创建基本的数组，以及数组如何映射为字符串。这个练习中我们会更加全面地展示数组和字符串的相似之处，并且深入了解更多内存布局的知识。

这个练习向你展示了C只是简单地将字符串储存为字符数组，并且在结尾加上 `'\0'` （空字符）。你可能在上个练习中得到了暗示，因为我们手动这样做了。下面我会通过将它与数字数组比较，用另一种方法更清楚地实现它。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int numbers[4] = {0};
    char name[4] = {'a'};

    // first, print them out raw
    printf("numbers: %d %d %d %d\n",
            numbers[0], numbers[1],
            numbers[2], numbers[3]);

    printf("name each: %c %c %c %c\n",
            name[0], name[1],
            name[2], name[3]);

    printf("name: %s\n", name);

    // setup the numbers
    numbers[0] = 1;
    numbers[1] = 2;
    numbers[2] = 3;
    numbers[3] = 4;

    // setup the name
    name[0] = 'Z';
    name[1] = 'e';
    name[2] = 'd';
    name[3] = '\0';

    // then print them out initialized
    printf("numbers: %d %d %d %d\n",
            numbers[0], numbers[1],
            numbers[2], numbers[3]);
```

```
    printf("name each: %c %c %c %c\n",
            name[0], name[1],
            name[2], name[3]);

    // print the name like a string
    printf("name: %s\n", name);

    // another way to use name
    char *another = "Zed";

    printf("another: %s\n", another);

    printf("another each: %c %c %c %c\n",
            another[0], another[1],
            another[2], another[3]);

    return 0;
}
```

在这段代码中，我们创建了一些数组，并对数组元素赋值。在 `numbers` 中我们设置了一些数字，然而在 `names` 中我们实际上手动构造了一个字符串。

你会看到什么

当你运行这段代码的时候，你应该首先看到所打印的数组的内容初始化为0值，之后打印初始化后的内容：

```
$ make ex9
cc -Wall -g    ex9.c   -o ex9
$ ./ex9
numbers: 0 0 0 0
name each: a
name: a
numbers: 1 2 3 4
name each: Z e d
name: Zed
another: Zed
another each: Z e d
$
```

你会注意到这个程序中有一些很有趣的事情：

- 我并没有提供全部的4个参数来初始化它。这是C的一个简写，如果你只提供了一个元素，剩下的都会为0。
- `numbers` 的每个元素被打印时，它们都输出0。
- `names` 的每个元素被打印时，只显示了第一个元素 `'a'` ，因为 `'\0'` 是特殊字符而不会显示。
- 然后我们首次打印 `names` ，打印出了 `"a"` ，因为在初始化表达式

中， `'a'` 字符之后的空间都用 `'\0'` 填充，是以 `'\0'` 结尾的有效字符串。
- 我们接着通过手动为每个元素赋值来建立数组，并且再次把它打印出来。看看他们发生了什么改变。现在 `numbers` 已经设置好了，看看 `names` 字符串如何正确打印出我的名字。
- 创建一个字符串也有两种语法：第六行的 `char name[4] = {'a'}` ，或者第44行的 `char *another = "name"` 。前者不怎么常用，你应该将后者用于字符串字面值。

注意我使用了相同的语法和代码风格来和整数数组和字符数组交互，但是 `printf` 认为 `name` 是个字符串。再次强调，这是因为对C语言来说，字符数组和字符串没有什么不同。

最后，当你使用字符串字面值时你应该用 `char *another = "Literal"` 语法，它会产生相同的东西，但是更加符合语言习惯，也更省事。

如何使它崩溃

C中所有bug的大多数来源都是忘了预留出足够的空间，或者忘了在字符串末尾加上一个 `'\0'` 。事实上，这些bug是非常普遍并且难以改正的，大部分优秀的C代码都不会使用C风格字符串。下一个练习中我们会学到如何彻底避免C风格字符串。

使这个程序崩溃的的关键就是拿掉字符串结尾的 `'\0'` 。下面是实现它的一些途径：

- 删掉 `name` 的初始化表达式。
- 无意中设置 `name[3] = 'A'` ，于是它就没有终止字符了。
- 将初始化表达式设置为 `{'a','a','a','a'}` ，于是就有过多的 `'a'` 字符，没有办法给 `'\0'` 留出位置。

试着想出一些其它的办法让它崩溃，并且在 `Valgrind` 下像往常一样运行这个程序，你可以看到具体发生了什么，以及错误叫什么名字。有时 `Valgrind` 并不能发现你犯的错误，则需要移动声明这些变量的地方看看是否能找出错误。这是C的黑魔法的一部分，有时变量的位置会改变bug。

附加题

- 将一些字符赋给 `numbers` 的元素，之后用 `printf` 一次打印一个字符，你会得到什么编译器警告？
- 对 `names` 执行上述的相反操作，把 `names` 当成 `int` 数组，并一次打印一个 `int` ， `Valgrind` 会提示什么？
- 有多少种其它的方式可以用来打印它？
- 如果一个字符数组占四个字节，一个整数也占4个字节，你可以像整数一样使用整个 `name` 吗？你如何用黑魔法实现它？
- 拿出一张纸，将每个数组画成一排方框，之后在纸上画出代码中的操作，看看是否正确。
- 将 `name` 转换成 `another` 的形式，看看代码是否能正常工作。

练习10：字符串数组和循环

> 原文：Exercise 10: Arrays Of Strings, Looping
>
> 译者：飞龙

你现在可以创建不同类型的数组，并且也知道了“字符串”和“字节数组”是相同的东西。接下来，我们要更进一步，创建一个包含字符串的数组。我也会介绍第一个循环结构，`for`循环来帮我们打印出这一新的数据结构。

这一章的有趣之处就是你的程序中已经有一个现成的字符串数组，`main`函数参数中的`char *argv[]`。下面这段代码打印出了所有你传入的命令行参数：

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;

    // go through each string in argv
    // why am I skipping argv[0]?
    for(i = 1; i < argc; i++) {
        printf("arg %d: %s\n", i, argv[i]);
    }

    // let's make our own array of strings
    char *states[] = {
        "California", "Oregon",
        "Washington", "Texas"
    };
    int num_states = 4;

    for(i = 0; i < num_states; i++) {
        printf("state %d: %s\n", i, states[i]);
    }

    return 0;
}
```

`for`循环的格式是这样的：

```c
for(INITIALIZER; TEST; INCREMENTER) {
    CODE;
}
```

下面是`for`循环的工作机制：

- `INITIALIZER` 中是用来初始化循环的代码，这个例子中它是 `i = 0` 。
- 接下来会检查 `TEST` 布尔表达式，如果为 `false` （0）则跳过 `CODE` ，不做任何事情。
- 执行 `CODE` ，做它要做的任何事情。
- 在 `CODE` 执行之后会执行 `INCREMENTER` 部分，通常情况会增加一些东西，比如这个例子是 `i++` 。
- 然后跳到第二步继续执行，直到 `TEST` 为 `false` （0）为止。

例子中的 `for` 循环使用 `argc` 和 `argv` ，遍历了命令行参数，像这样：

- OS将每个命令行参数作为字符串传入 `argv` 数组，程序名称 `./ex10` 在下标为0的位置，剩余的参数紧随其后。
- OS将 `argc` 置为 `argv` 数组中参数的数量，所以你可以遍历它们而不会越界。要记住如果你提供了一个参数，程序名称是第一个，参数应该在第二个。
- 接下来程序使用 `i < argc` 测试 `i` 是否使用 `argc` ，由于最开始 `1 < 2` ，测试通过。
- 之后它会执行代码，输出 `i` ，并且将 `i` 用做 `argv` 的下标。
- 然后使用 `i++` 来运行自增语句，它是 `i = i + 1` 的便捷形式。
- 程序一直重复上面的步骤，直到 `i < argc` 值为 `false` （0），这时退出循环但程序仍然继续执行。

你会看到什么

你需要用两种方法运行它来玩转这个程序。第一种方法是向命令行参数传递一些东西来设置 `argc` 和 `argv` 。第二种是不传入任何参数，于是你可以看到第一次的 `for` 循环没有被执行，由于 `i < argc` 值为 `false` 。

理解字符串数组

你应该可以从这个练习中弄明白，你在C语言中通过混合 `char *str = "blah"` 和 `char str[] = {'b','l','a','h'}` 语法构建二维数组来构建字符串数组。第十四行的 `char *states[] = {...}` 语法就是这样的二维混合结构，其中每个字符串都是数组的一个元素，字符串的每个字符又是字符串的一个元素。

感到困惑吗？多维的概念是很多人从来都不会去想的，所以你应该在纸上构建这一字符串数组：

- 在纸的左边为每个字符串画一个小方格，带有它们的下标。
- 然后在方格上方写上每个字符的下标。
- 接着将字符串中的字符填充到方格内。
- 画完之后，在纸上模拟代码的执行过程。

理解它的另一种方法是在你熟悉的语言，比如Python或Ruby中构建相同的结构。

如何使它崩溃

- 使用你喜欢的另一种语言，来写这个程序。传入尽可能多的命令行参数，看看是否能通过传入过多参数使其崩溃。
- 将 `i` 初始化为0看看会发生什么。是否也需要改动 `argc` ，不改动的话它能正常工作吗？为什么下标从0开始可以正常工作？
- 将 `num_states` 改为错误的值使它变大，来看看会发生什么。

附加题

- 弄清楚在 `for` 循环的每一部分你都可以放置什么样的代码。
- 查询如何使用 `','` （逗号）字符来在 `for` 循环的每一部分中， `';'` （分号）之间分隔多条语句。
- 查询 `NULL` 是什么东西，尝试将它用做 `states` 的一个元素，看看它会打印出什么。
- 看看你是否能在打印之前将 `states` 的一个元素赋值给 `argv` 中的元素，再试试相反的操作。

练习11：While循环和布尔表达式

> 原文：Exercise 11: While-Loop And Boolean Expressions
>
> 译者：飞龙

你已经初步了解C语言如何处理循环，但是你可能不是很清楚布尔表达式 `i < argc` 是什么。在学习 `while` 循环之前，让我先来对布尔表达式做一些解释。

在C语言中，实际上没有真正的“布尔”类型，而是用一个整数来代替，0代表 `false` ，其它值代表 `true` 。上一个练习中表达式 `i < argc` 实际上值为1或者0，并不像Python是显式的 `Ture` 或者 `False` 。这是C语言更接近计算机工作方式的另一个例子，因为计算机只把值当成数字。

现在用 `while` 循环来实现和上一个练习相同的函数。这会让你使用两种循环，来观察两种循环是什么关系。

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    // go through each string in argv

    int i = 0;
    while(i < argc) {
        printf("arg %d: %s\n", i, argv[i]);
        i++;
    }

    // let's make our own array of strings
    char *states[] = {
        "California", "Oregon",
        "Washington", "Texas"
    };

    int num_states = 4;
    i = 0;  // watch for this
    while(i < num_states) {
        printf("state %d: %s\n", i, states[i]);
        i++;
    }

    return 0;
}
```

你可以看到 `while` 循环的语法更加简单：

```
while(TEST) {
    CODE;
}
```

只要 `TEST` 为 `true` （非0），就会一直运行 `CODE` 中的代码。这意味着如果要达到和 `for` 循环同样的效果，我们需要自己写初始化语句，以及自己来使 `i` 增加。

你会看到什么

输出基本相同，所以我做了一点修改，使你可以看到它运行的另一种方式。

```
$ make ex11
cc -Wall -g    ex11.c   -o ex11
$ ./ex11
arg 0: ./ex11
state 0: California
state 1: Oregon
state 2: Washington
state 3: Texas
$
$ ./ex11 test it
arg 0: ./ex11
arg 1: test
arg 2: it
state 0: California
state 1: Oregon
state 2: Washington
state 3: Texas
$
```

如何使它崩溃

在你自己的代码中，应优先选择 `for` 循环而不是 `while` 循环，因为 `for` 循环不容易崩溃。下面是几点普遍的原因：

- 忘记初始化 `int i` ，使循环发生错误。
- 忘记初始化第二个循环的 `i` ，于是 `i` 还保留着第一个循环结束时的值。你的第二个循环可能执行也可能不会执行。
- 忘记在最后执行 `i++` 自增，你会得到一个“死循环”，它是在你开始编程的第一个或前两个十年中，最可怕的问题之一。

附加题

- 让这些循环倒序执行，通过使用 `i--` 从 `argc` 开始递减直到0。你可能需要

做一些算数操作让数组的下标正常工作。
* 使用 `while` 循环将 `argv` 中的值复制到 `states` 。
* 让这个复制循环不会执行失败，即使 `argv` 之中有很多元素也不会全部放进 `states` 。
* 研究你是否真正复制了这些字符串。答案可能会让你感到意外和困惑。

练习12：If，Else If，Else

> 原文：Exercise 12: If, Else-If, Else
>
> 译者：飞龙

`if` 语句是每个编程语言中共有的特性，包括C语言。下面是一段代码，使用了 `if` 语句来确保只传入了一个或两个命令行参数：

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;

    if(argc == 1) {
        printf("You only have one argument. You suck.\n");
    } else if(argc > 1 && argc < 4) {
        printf("Here's your arguments:\n");

        for(i = 0; i < argc; i++) {
            printf("%s ", argv[i]);
        }
        printf("\n");
    } else {
        printf("You have too many arguments. You suck.\n");
    }

    return 0;
}
```

`if` 语句的格式为：

```c
if(TEST) {
    CODE;
} else if(TEST) {
    CODE;
} else {
    CODE;
}
```

下面是其它语言和C的差异：

- 像之前提到的那样，`TEST` 表达式值为0时为 `false`，其它情况为 `true`。
- 你需要在 `TEST` 周围写上圆括号，其它语言可能不用。
- （只有单条语句时）你并不需要使用花括号 `{}` 来闭合代码，但是这是一种非

常不好的格式，不要这么写。花括号让一个分支的代码的开始和结束变得清晰。如果你不把代码写在里面会出现错误。

除了上面那些，就和其它语言一样了。 `else if` 或者 `else` 的部分并不必须出现。

你会看到什么

这段代码非常易于运行和尝试：

```
$ make ex12
cc -Wall -g    ex12.c   -o ex12
$ ./ex12
You only have one argument. You suck.
$ ./ex12 one
Here's your arguments:
./ex12 one
$ ./ex12 one two
Here's your arguments:
./ex12 one two
$ ./ex12 one two three
You have too many arguments. You suck.
$
```

如何使它崩溃

使这段代码崩溃并不容易，因为它太简单了。尝试把 `if` 语句的测试表达式搞乱：

- 移除 `else` 部分，使它不能处理边界情况。
- 将 `&&` 改为 `||` ，于是你会把“与”操作变成“或”操作，并且看看会发生什么。

附加题

- 我已经向你简短地介绍了 `&&` ，它执行“与”操作。上网搜索与之不同的“布尔运算符”。
- 为这个程序编写更多的测试用例，看看你会写出什么。
- 回到练习10和11，使用 `if` 语句使循环提前退出。你需要 `break` 语句来实现它，搜索它的有关资料。
- 第一个判断所输出的话真的正确吗？由于你的“第一个参数”不是用户输入的第一个参数，把它改正。

练习13：Switch语句

> 原文：Exercise 13: Switch Statement
>
> 译者：飞龙

在其它类似Ruby的语言中， `switch` 语句可以处理任意类型的表达式。一些语言比如Python没有 `switch` 语句，因为带有布尔表达式的 `if` 语句可以做相同的事情。对于这些语言， `switch` 语句比 `if` 语句更加灵活，然而内部的机制是一样的。

C中的 `switch` 语句与它们不同，实际上是一个“跳转表”。你只能够放置结果为整数的表达式，而不是一些随机的布尔表达式，这些整数用于计算从 `swicth` 顶部到匹配部分的跳转。下面有一段代码，我要分解它来让你理解“跳转表”的概念：

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    if(argc != 2) {
        printf("ERROR: You need one argument.\n");
        // this is how you abort a program
        return 1;
    }

    int i = 0;
    for(i = 0; argv[1][i] != '\0'; i++) {
        char letter = argv[1][i];

        switch(letter) {
            case 'a':
            case 'A':
                printf("%d: 'A'\n", i);
                break;

            case 'e':
            case 'E':
                printf("%d: 'E'\n", i);
                break;

            case 'i':
            case 'I':
                printf("%d: 'I'\n", i);
                break;

            case 'o':
            case 'O':
                printf("%d: 'O'\n", i);
                break;
```

```
            case 'u':
            case 'U':
                printf("%d: 'U'\n", i);
                break;

            case 'y':
            case 'Y':
                if(i > 2) {
                    // it's only sometimes Y
                    printf("%d: 'Y'\n", i);
                }
                break;

            default:
                printf("%d: %c is not a vowel\n", i, letter);
        }
    }

    return 0;
}
```

在这个程序中我们接受了单一的命令行参数，并且用一种极其复杂的方式打印出所有原因，来向你演示 `switch` 语句。下面是 `swicth` 语句的工作原理：

- 编译器会标记 `swicth` 语句的顶端，我们先把它记为地址Y。
- 接着对 `switch` 中的表达式求值，产生一个数字。在上面的例子中，数字为 `argv[1]` 中字母的原始的ASCLL码。
- 编译器也会把每个类似 `case 'A'` 的 `case` 代码块翻译成这个程序中距离语句顶端的地址，所以 `case 'A'` 就在 `Y + 'A'` 处。
- 接着计算是否 `Y+letter` 位于 `switch` 语句中，如果距离太远则会将其调整为 `Y+Default` 。
- 一旦计算出了地址，程序就会“跳”到代码的那个位置并继续执行。这就是一些 `case` 代码块中有 `break` 而另外一些没有的原因。
- 如果输出了 `'a'` ，那它就会跳到 `case 'a'` ，它里面没有 `break` 语句，所以它会贯穿执行底下带有代码和 `break` 的 `case 'A'` 。
- 最后它执行这段代码，执行 `break` 完全跳出 `switch` 语句块。

> 译者注：更常见的情况是，gcc会在空白处单独构建一张跳转表，各个偏移处存放对应的 `case` 语句的地址。Y不是 `switch` 语句的起始地址，而是这张表的起始地址。程序会跳转到 `*(Y + 'A')` 而不是 `Y + 'A'` 处。

这是对 `swicth` 语句工作原理的一个深究，然而实际操作中你只需要记住下面几条简单的原则：

- 总是要包含一个 `default:` 分支，可以让你接住被忽略的输入。
- 不要允许“贯穿”执行，除非你真的想这么做，这种情况下最好添加一个 `//fallthrough` 的注释。
- 一定要先编写 `case` 和 `break` ，再编写其中的代码。

- 如果能够简化的话，用 `if` 语句代替。

你会看到什么

下面是我运行它的一个例子，也演示了传入命令行参数的不同方法：

```
$ make ex13
cc -Wall -g    ex13.c   -o ex13
$ ./ex13
ERROR: You need one argument.
$
$ ./ex13 Zed
0: Z is not a vowel
1: 'E'
2: d is not a vowel
$
$ ./ex13 Zed Shaw
ERROR: You need one argument.
$
$ ./ex13 "Zed Shaw"
0: Z is not a vowel
1: 'E'
2: d is not a vowel
3:   is not a vowel
4: S is not a vowel
5: h is not a vowel
6: 'A'
7: w is not a vowel
$
```

记住在代码的开始有个 `if` 语句，当没有提供足够的参数时使用 `return 1` 返回。返回非0是你提示操作系统程序出错的办法。任何大于0的值都可以在脚本中测试，其它程序会由此知道发生了什么。

如何使它崩溃

破坏一个 `switch` 语句块太容易了。下面是一些方法，你可以挑一个来用：

- 忘记写 `break` ，程序就会运行两个或多个代码块，这些都是你不想运行的。
- 忘记写 `default` ，程序会在静默中忽略你所忘记的值。
- 无意中将一些带有预料之外的值的变量放入 `switch` 中，比如带有奇怪的值的 `int` 。
- 在 `switch` 中是否未初始化的值。

你也可以使用一些别的方法使这个程序崩溃。试着看你能不能自己做到它。

附加题

- 编写另一个程序，在字母上做算术运算将它们转换为小写，并且在`switch`中移除所有额外的大写字母。
- 使用`','`（逗号）在`for`循环中初始化`letter`。
- 使用另一个`for`循环来让它处理你传入的所有命令行参数。
- 将这个`switch`语句转为`if`语句，你更喜欢哪个呢？
- 在“Y”的例子中，我在`if`代码块外面写了个`break`。这样会产生什么效果？如果把它移进`if`代码块，会发生什么？自己试着解答它，并证明你是正确的。

练习14：编写并使用函数

> 原文：Exercise 14: Writing And Using Functions
>
> 译者：飞龙

到现在为止，你只使用了作为`stdio.h`头文件一部分的函数。在这个练习中你将要编写并使用自己的函数。

```c
#include <stdio.h>
#include <ctype.h>

// forward declarations
int can_print_it(char ch);
void print_letters(char arg[]);

void print_arguments(int argc, char *argv[])
{
    int i = 0;

    for(i = 0; i < argc; i++) {
        print_letters(argv[i]);
    }
}

void print_letters(char arg[])
{
    int i = 0;

    for(i = 0; arg[i] != '\0'; i++) {
        char ch = arg[i];

        if(can_print_it(ch)) {
            printf("'%c' == %d ", ch, ch);
        }
    }

    printf("\n");
}

int can_print_it(char ch)
{
    return isalpha(ch) || isblank(ch);
}


int main(int argc, char *argv[])
{
    print_arguments(argc, argv);
    return 0;
}
```

在这个例子中你创建了函数来打印任何属于“字母”和“空白”的字符。下面是一个分解：

ex14.c:2

包含了新的头文件，所以你可以访问 `isalpha` 和 `isblank` 。

ex14.c:5-6

告诉C语言你稍后会在你的程序中使用一些函数，它们实际上并没有被定义。这叫做“前向声明”，它解决了要想使用函数先要定义的鸡和蛋的问题。

ex14.c:8-15

定义 `print_arguments` ，它知道如何打印通常由 `main` 函数获得的相同字符串数组。

ex14.c:17-30

定义了 `can_print_it` ，它只是简单地将 `isalpha(ch) || isblank(ch)` 的真值（0或1）返回给它的调用者 `print_letters` 。

ex14.c:38-42

最后 `main` 函数简单地调用 `print_arguments` ，来启动整个函数链。

我不应该描述每个函数里都有什么，因为这些都是你之前遇到过的东西。你应该看到的是，我只是像你定义 `main` 函数一样来定义其它函数。唯一的不同就是如果你打算使用当前文件中没有碰到过的函数，你应该事先告诉C。这就是代码顶部的“前向声明”的作用。

你会看到什么

向这个程序传入不同的命令行参数来玩转它，这样会遍历你函数中的所有路径。这里演示了我和它的交互：

```
$ make ex14
cc -Wall -g    ex14.c   -o ex14

$ ./ex14
'e' == 101 'x' == 120

$ ./ex14 hi this is cool
'e' == 101 'x' == 120
'h' == 104 'i' == 105
't' == 116 'h' == 104 'i' == 105 's' == 115
'i' == 105 's' == 115
'c' == 99 'o' == 111 'o' == 111 'l' == 108

$ ./ex14 "I go 3 spaces"
'e' == 101 'x' == 120
'I' == 73 ' ' == 32 'g' == 103 'o' == 111 ' ' == 32 ' ' == 32 's'
 == 115 'p' == 112 'a' == 97 'c' == 99 'e' == 101 's' == 115
$
```

`isalpha` 和 `isblank` 做了检查提供的字符是否是字母或者空白字符的所有工作。当我最后一次运行时，它打印出除了 `'3'` 之外的任何东西，因为它是一个数字。

如何使它崩溃

下面是使它崩溃的两种不同的方法：

- 通过移除前向声明来把编译器搞晕。它会报告 `can_print_it` 和 `print_letters` 的错误。
- 当你在 `main` 中调用 `print_arguments` 时，试着使 `argc` 加1，于是它会越过 `argv` 数组的最后一个元素。

附加题

- 重新编写这些函数，使它们的数量减少。比如，你真的需要 `can_print_it` 吗？
- 使用 `strlen` 函数，让 `print_arguments` 知道每个字符串参数都有多长，之后将长度传入 `print_letters` 。然后重写 `print_letters` ，让它只处理固定的长度，不按照 `'\0'` 终止符。你需要 `#include <string.h>` 来实现它。
- 使用 `man` 来查询 `isalpha` 和 `isblank` 的信息。使用其它相似的函数来只打印出数字或者其它字符。
- 上网浏览不同的人喜欢什么样的函数格式。永远不要使用“K&R”语法，因为它过时了，而且容易使人混乱，但是当你碰到一些人使用这种格式时，要理解代码做了什么。

练习15：指针，可怕的指针

> 原文：Exercise 15: Pointers Dreaded Pointers
>
> 译者：飞龙

指针是C中的一个著名的谜之特性，我会试着通过教授你一些用于处理它们的词汇，使之去神秘化。指针实际上并不复杂，只不过它们经常以一些奇怪的方式被滥用，这样使它们变得难以使用。如果你避免这些愚蠢的方法来使用指针，你会发现它们难以置信的简单。

要想以一种我们可以谈论的方式来讲解指针，我会编写一个无意义的程序，它以三种方式打印了一组人的年龄：

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    // create two arrays we care about
    int ages[] = {23, 43, 12, 89, 2};
    char *names[] = {
        "Alan", "Frank",
        "Mary", "John", "Lisa"
    };

    // safely get the size of ages
    int count = sizeof(ages) / sizeof(int);
    int i = 0;

    // first way using indexing
    for(i = 0; i < count; i++) {
        printf("%s has %d years alive.\n",
                names[i], ages[i]);
    }

    printf("---\n");

    // setup the pointers to the start of the arrays
    int *cur_age = ages;
    char **cur_name = names;

    // second way using pointers
    for(i = 0; i < count; i++) {
        printf("%s is %d years old.\n",
                *(cur_name+i), *(cur_age+i));
    }

    printf("---\n");
```

```
    // third way, pointers are just arrays
    for(i = 0; i < count; i++) {
        printf("%s is %d years old again.\n",
                cur_name[i], cur_age[i]);
    }

    printf("---\n");

    // fourth way with pointers in a stupid complex way
    for(cur_name = names, cur_age = ages;
            (cur_age - ages) < count;
            cur_name++, cur_age++)
    {
        printf("%s lived %d years so far.\n",
                *cur_name, *cur_age);
    }

    return 0;
}
```

在解释指针如何工作之前，让我们逐行分解这个程序，这样你可以对发生了什么有所了解。当你浏览这个详细说明时，试着自己在纸上回答问题，之后看看你猜测的结果符合我对指针的描述。

ex15.c:6-10

创建了两个数组，`ages` 储存了一些 `int` 数据，`names` 储存了一个字符串数组。

ex15.c:12-13

为之后的 `for` 循环创建了一些变量。

ex15.c:16-19

你知道这只是遍历了两个数组，并且打印出每个人的年龄。它使用了 `i` 来对数组索引。

ex15.c:24

创建了一个指向 `ages` 的指针。注意 `int *` 创建“指向整数的指针”的指针类型的用法。它很像 `char *` ，意义是“指向字符的指针”，而且字符串是字符的数组。是不是很相似呢？

ex15.c:25

创建了指向 `names` 的指针。`char *` 已经是“指向 `char` 的指针”了，所以它只是个字符串。你需要两个层级，因为 `names` 是二维的，也就是说你需要 `char **` 作为“指向‘指向字符的指针’的指针”。把它学会，并且自己解释它。

ex15.c:28-31

遍历 `ages` 和 `names` ，但是使用“指针加偏移 `i` ”。 `*(cur_name+i)` 和 `name[i]` 是一样的，你应该把它读作“ `cur_name` 指针加 `i` ’的值”。

ex15.c:35-39

这里展示了访问数组元素的语法和指针是相同的。

ex15.c:44-50

另一个十分愚蠢的循环和其它两个循环做着相同的事情，但是它用了各种指针算术运算来代替：

ex15.c:44

通过将 `cur_name` 和 `cur_age` 置为 `names` 和 `age` 数组的起始位置来初始化 `for` 循环。

ex15.c:45

`for` 循环的测试部分比较 `cur_age` 指针和 `ages` 起始位置的距离，为什么可以这样写呢？

ex15.c:46

`for` 循环的增加部分增加了 `cur_name` 和 `cur_age` 的值，这样它们可以只想 `names` 和 `ages` 的下一个元素。

ex15.c:48-49

`cur_name` 和 `cur_age` 的值现在指向了相应数组中的一个元素，我们我可以通过 `*cur_name` 和 `*cur_age` 来打印它们，这里的意思是“ `cur_name` 和 `cur_age` 指向的值”。

这个看似简单的程序却包含了大量的信息，其目的是在我向你讲解之前尝试让你自己弄清楚指针。直到你写下你认为指针做了什么之前，不要往下阅读。

你会看到什么

在你运行这个程序之后，尝试根据打印出的每一行追溯到代码中产生它们的那一行。在必要情况下，修改 `printf` 调用来确认你得到了正确的行号：

```
$ make ex15
cc -Wall -g    ex15.c   -o ex15
$ ./ex15
Alan has 23 years alive.
Frank has 43 years alive.
Mary has 12 years alive.
John has 89 years alive.
Lisa has 2 years alive.
---
Alan is 23 years old.
Frank is 43 years old.
Mary is 12 years old.
John is 89 years old.
Lisa is 2 years old.
---
Alan is 23 years old again.
Frank is 43 years old again.
Mary is 12 years old again.
John is 89 years old again.
Lisa is 2 years old again.
---
Alan lived 23 years so far.
Frank lived 43 years so far.
Mary lived 12 years so far.
John lived 89 years so far.
Lisa lived 2 years so far.
$
```

解释指针

当你写下一些类似`ages[i]`的东西时，你实际上在用`i`中的数字来索引`ages`。如果`i`的值为0，那么就等同于写下`ages[0]`。我们把`i`叫做下标，因为它是`ages`中的一个位置。它也能称为地址，这是“我想要`ages`位于地址`i`处的整数”中的说法。

如果`i`是个下标，那么`ages`又是什么？对C来说`ages`是在计算机中那些整数的起始位置。当然它也是个地址，C编译器会把任何你键入`ages`的地方替换为数组中第一个整数的地址。另一个理解它的办法就是把`ages`当作“数组内部第一个整数的地址”，但是它是整个计算机中的地址，而不是像`i`一样的`ages`中的地址。`ages`数组的名字在计算机中实际上是个地址。

这就产生了一种特定的实现：C把你的计算机看成一个庞大的字节数组。显然这样不会有什么用处，于是C就在它的基础上构建出类型和大小的概念。你已经在前面的练习中看到了它是如何工作的，但现在你可以开始了解C对你的数组做了下面一些事情：

- 在你的计算机中开辟一块内存。
- 将`ages`这个名字“指向”它的起始位置。

- 通过选取 `ages` 作为基址，并且获取位置为 `i` 的元素，来对内存块进行索引。
- 将 `ages+i` 处的元素转换成大小正确的有效的 `int` ，这样就返回了你想要的结果：下标 `i` 处的 `int` 。

如果你可以选取 `ages` 作为基址，之后加上比如 `i` 的另一个地址，你是否就能随时构造出指向这一地址的指针呢？是的，这种东西就叫做指针。这也是 `cur_age` 和 `cur_name` 所做的事情，它们是指向计算机中这一位置的变量， `ages` 和 `names` 就处于这一位置。之后，示例程序移动它们，或者做了一些算数运算，来从内存中获取值。在其中一个实例中，只是简单地将 `cur_age` 加上 `i` ，这样等同于 `array[i]` 。在最后一个 `for` 循环中，这两个指针在没有 `i` 辅助的情况下自己移动，被当做数组基址和整数偏移合并到一起的组合。

指针仅仅是指向计算机中的某个地址，并带有类型限定符，所以你可以通过它得到正确大小的数据。它类似于将 `ages` 和 `i` 组合为一个数据类型的东西。C了解指针指向什么地方，所指向的数据类型，这些类型的大小，以及如何为你获取数据。你可以像 `i` 一样增加它们，减少它们，对他们做加减运算。然而它们也像是 `ages` ，你可以通过它获取值，放入新的值，或执行全部的数组操作。

指针的用途就是让你手动对内存块进行索引，一些情况下数组并不能做到。绝大多数情况中，你可能打算使用数组，但是一些处理原始内存块的情况，是指针的用武之地。指针向你提供了原始的、直接的内存块访问途径，让你能够处理它们。

在这一阶段需要掌握的最后一件事，就是你可以对数组和指针操作混用它们绝大多数的语法。你可以对一个指针使用数组的语法来访问指向的东西，也可以对数组的名字做指针的算数运算。

实用的指针用法

你可以用指针做下面四个最基本的操作：

- 向OS申请一块内存，并且用指针处理它。这包括字符串，和一些你从来没见过的东西，比如结构体。
- 通过指针向函数传递大块的内存（比如很大的结构体），这样不必把全部数据都传递进去。
- 获取函数的地址用于动态调用。
- 对一块内存做复杂的搜索，比如，转换网络套接字中的字节，或者解析文件。

对于你看到的其它所有情况，实际上应当使用数组。在早期，由于编译器不擅长优化数组，人们使用指针来加速它们的程序。然而，现在访问数组和指针的语法都会翻译成相同的机器码，并且表现一致。由此，你应该每次尽可能使用数组，并且按需将指针用作提升性能的手段。

指针词库

现在我打算向你提供一个词库，用于读写指针。当你遇到复杂的指针语句时，试着参考它并且逐字拆分语句（或者不要使用这个语句，因为有可能并不好）：

`type *ptr`

`type` 类型的指针，名为 `ptr` 。

`*ptr`

`ptr` 所指向位置的值。

`*(ptr + i)`

（ `ptr` 所指向位置加上 `i` ）的值。

> 译者注：以字节为单位的话，应该是 `ptr` 所指向的位置再加上 `sizeof(type) * i` 。

`&thing`

`thing` 的地址。

`type *ptr = &thing`

名为 `ptr` ， `type` 类型的指针，值设置为 `thing` 的地址。

`ptr++`

自增 `ptr` 指向的位置。

我们将会使用这份简单的词库来拆解这本书中所有的指针用例。

指针并不是数组

无论怎么样，你都不应该把指针和数组混为一谈。它们并不是相同的东西，即使C让你以一些相同的方法来使用它们。例如，如果你访问上面代码中的 `sizeof(cur_age)` ，你会得到指针的大小，而不是它指向数组的大小。如果你想得到整个数组的大小，你应该使用数组的名称 `age` ，就像第12行那样。

> 译者注，除了 `sizeof` 、 `&` 操作和声明之外，数组名称都会被编译器推导为指向其首个元素的指针。对于这些情况，不要用“是”这个词，而是要用“推导”。

如何使它崩溃

你可以通过将指针指向错误的位置来使程序崩溃：

- 试着将 `cur_age` 指向 `names` 。可以需要C风格转换来强制执行，试着查阅相关资料把它弄明白。
- 在最后的 `for` 循环中，用一些古怪的方式使计算发生错误。
- 试着重写循环，让它们从数组的最后一个元素开始遍历到首个元素。这比看上去要困难。

附加题

- 使用访问指针的方式重写所有使用数组的地方。
- 使用访问数组的方式重写所有使用指针的地方。
- 在其它程序中使用指针来代替数组访问。
- 使用指针来处理命令行参数，就像处理`names`那样。
- 将获取值和获取地址组合到一起。
- 在程序末尾添加一个`for`循环，打印出这些指针所指向的地址。你需要在`printf`中使用`%p`。
- 对于每一种打印数组的方法，使用函数来重写程序。试着向函数传递指针来处理数据。记住你可以声明接受指针的函数，但是可以像数组那样用它。
- 将`for`循环改为`while`循环，并且观察对于每种指针用法哪种循环更方便。

练习16：结构体和指向它们的指针

> 原文：Exercise 16: Structs And Pointers To Them
>
> 译者：飞龙

在这个练习中你将会学到如何创建`struct`，将一个指针指向它们，以及使用它们来理解内存的内部结构。我也会借助上一节课中的指针知识，并且让你使用`malloc`从原始内存中构造这些结构体。

像往常一样，下面是我们将要讨论的程序，你应该把它打下来并且使它正常工作：

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

struct Person {
    char *name;
    int age;
    int height;
    int weight;
};

struct Person *Person_create(char *name, int age, int height, int weight)
{
    struct Person *who = malloc(sizeof(struct Person));
    assert(who != NULL);

    who->name = strdup(name);
    who->age = age;
    who->height = height;
    who->weight = weight;

    return who;
}

void Person_destroy(struct Person *who)
{
    assert(who != NULL);

    free(who->name);
    free(who);
}

void Person_print(struct Person *who)
{
    printf("Name: %s\n", who->name);
```

```
    printf("\tAge: %d\n", who->age);
    printf("\tHeight: %d\n", who->height);
    printf("\tWeight: %d\n", who->weight);
}

int main(int argc, char *argv[])
{
    // make two people structures
    struct Person *joe = Person_create(
            "Joe Alex", 32, 64, 140);

    struct Person *frank = Person_create(
            "Frank Blank", 20, 72, 180);

    // print them out and where they are in memory
    printf("Joe is at memory location %p:\n", joe);
    Person_print(joe);

    printf("Frank is at memory location %p:\n", frank);
    Person_print(frank);

    // make everyone age 20 years and print them again
    joe->age += 20;
    joe->height -= 2;
    joe->weight += 40;
    Person_print(joe);

    frank->age += 20;
    frank->weight += 20;
    Person_print(frank);

    // destroy them both so we clean up
    Person_destroy(joe);
    Person_destroy(frank);

    return 0;
}
```

我打算使用一种和之前不一样的方法来描述这段程序。我并不会对程序做逐行的拆分，而是由你自己写出来。我会基于程序所包含的部分来给你提示，你的任务就是写出每行是干什么的。

包含（`include`）

我包含了一些新的头文件，来访问一些新的函数。每个头文件都提供了什么东西？

`struct Person`

这就是我创建结构体的地方了，结构体含有四个成员来描述一个人。最后我们得到了一个复合类型，让我们通过一个名字来整体引用这些成员，或它们的每一个。这就像数据库表中的一行或者OOP语言中的一个类那样。

`Pearson_create` 函数

我需要一个方法来创建这些结构体，于是我定义了一个函数来实现。下面是这个函数做的几件重要的事情：

- 使用用于内存分配的 `malloc` 来向OS申请一块原始的内存。
- 向 `malloc` 传递 `sizeof(struct Person)` 参数，它计算结构体的大小，包含其中的所有成员。
- 使用了 `assert` 来确保从 `malloc` 得到一块有效的内存。有一个特殊的常亮叫做 `NULL` ，表示“未设置或无效的指针”。这个 `assert` 大致检查了 `malloc` 是否会返回 `NULL` 。
- 使用 `x->y` 语法来初始化 `struct Person` 的每个成员，它指明了所初始化的成员。
- 使用 `strdup` 来复制字符串 `name` ，是为了确保结构体真正拥有它。 `strdup` 的行为实际上类似 `malloc` 但是它同时会将原来的字符串复制到新创建的内存。

> 译者注： `x->y` 是 `(*x).y` 的简写。

`Person_destroy` 函数

如果定义了创建函数，那么一定需要一个销毁函数，它会销毁 `Person` 结构体。我再一次使用了 `assert` 来确保不会得到错误的输入。接着我使用了 `free` 函数来交还通过 `malloc` 和 `strdup` 得到的内存。如果你不这么做则会出现“内存泄露”。

> 译者注：不想显式释放内存又能避免内存泄露的办法是引入 `libGC` 库。你需要把所有的 `malloc` 换成 `GC_malloc` ，然后把所有的 `free` 删掉。

`Person_print` 函数

接下来我需要一个方法来打印出人们的信息，这就是这个函数所做的事情。它用了相同的 `x->y` 语法从结构体中获取成员来打印。

`main` 函数

我在 `main` 函数中使用了所有前面的函数和 `struct Person` 来执行下面的事情：

- 创建了两个人： `joe` 和 `frank` 。
- 把它们打印出来，注意我用了 `%p` 占位符，所以你可以看到程序实际上把结构体放到了哪里。
- 把它们的年龄增加20岁，同时增加它们的体重。
- 之后打印出每个人。
- 最后销毁结构体，以正确的方式清理它们。

请仔细阅读上面的描述，然后做下面的事情：

- 查询每个你不了解的函数或头文件。记住你通常可以使用 `man 2 function` 或者 `man 3 function` 来让它告诉你。你也可以上网搜索资料。
- 在每一行上方编写注释，写下这一行代码做了什么。
- 跟踪每一个函数调用和变量，你会知道它在程序中是在哪里出现的。

- 同时也查询你不清楚的任何符号。

你会看到什么

在你使用描述性注释扩展程序之后，要确保它实际上能够运行，并且产生下面的输出：

```
$ make ex16
cc -Wall -g    ex16.c   -o ex16

$ ./ex16
Joe is at memory location 0xeba010:
Name: Joe Alex
    Age: 32
    Height: 64
    Weight: 140
Frank is at memory location 0xeba050:
Name: Frank Blank
   Age: 20
   Height: 72
   Weight: 180
Name: Joe Alex
   Age: 52
   Height: 62
   Weight: 180
Name: Frank Blank
   Age: 40
   Height: 72
   Weight: 200
```

解释结构体

如果你完成了我要求的任务，你应该理解了结构体。不过让我来做一个明确的解释，确保你真正理解了它。

C中的结构体是其它数据类型（变量）的一个集合，它们储存在一块内存中，然而你可以通过独立的名字来访问每个变量。它们就类似于数据库表中的一行记录，或者面向对象语言中的一个非常简单的类。让我们以这种方式来理解它：

- 在上面的代码中，你创建了一个结构体，它们的成员用于描述一个人：名称、年龄、体重、身高。
- 每个成员都有一个类型，比如是`int`。
- C会将它们打包到一起，于是它们可以用单个的结构体来存放。
- `struct Person`是一个复合类型，这意味着你可以在同种表达式中将其引用为其它的数据类型。
- 你可以将这一紧密的组合传递给其它函数，就像`Person_print`那样。
- 如果结构体是指针的形式，接着你可以使用`x->y`通过它们的名字来访问结构

体中独立的部分。
- 还有一种创建结构体的方法，不需要指针，通过 `x.y` 来访问。你将会在附加题里面见到它。

如果你不使用结构体，则需要自己计算出大小、打包以及定位出指定内容的内存片位置。实际上，在大多数早期（甚至现在的一些）的汇编代码中，这就是唯一的方式。在C中你就可以让C来处理这些复合数据类型的内存构造，并且专注于和它们交互。

如何使它崩溃

使这个程序崩溃的办法涉及到使用指针和 `malloc` 系统的方法：

- 试着传递 `NULL` 给 `Person_destroy` 来看看会发生什么。如果它没有崩溃，你必须移除Makefile的 `CFLAGS` 中的 `-g` 选项。
- 在结尾处忘记调用 `Person_destroy` ，在 `Valgrind` 下运行程序，你会看到它报告出你忘记释放内存。弄清楚你应该向 `valgrind` 传递什么参数来让它向你报告内存如何泄露。
- 忘记在 `Person_destroy` 中释放 `who->name` ，并且对比两次的输出。同时，使用正确的选项来让 `Valgrind` 告诉你哪里错了。
- 这一次，向 `Person_print` 传递 `NULL` ，并且观察 `Valgrind` 会输出什么。
- 你应该明白了 `NULL` 是个使程序崩溃的快速方法。

附加题

在这个练习的附加题中我想让你尝试一些有难度的东西：将这个程序改为不用指针和 `malloc` 的版本。这可能很困难，所以你需要研究下面这些东西：

- 如何在栈上创建结构体，就像你创建任何其它变量那样。
- 如何使用 `x.y` 而不是 `x->y` 来初始化结构体。
- 如何不使用指针来将结构体传给其它函数。

练习17：堆和栈的内存分配

> 原文：Exercise 17: Heap And Stack Memory Allocation
>
> 译者：飞龙

在这个练习中，你会在难度上做一个大的跳跃，并且创建出用于管理数据库的完整的小型系统。这个数据库并不实用也存储不了太多东西，然而它展示了大多数到目前为止你学到的东西。它也以更加正规的方法介绍了内存分配，以及带领你熟悉文件处理。我们使用了一些文件IO函数，但是我并不想过多解释它们，你可以先试着自己理解。

像通常一样，输入下面整个程序，并且使之正常工作，之后我们会进行讨论：

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#define MAX_DATA 512
#define MAX_ROWS 100

struct Address {
    int id;
    int set;
    char name[MAX_DATA];
    char email[MAX_DATA];
};

struct Database {
    struct Address rows[MAX_ROWS];
};

struct Connection {
    FILE *file;
    struct Database *db;
};

void die(const char *message)
{
    if(errno) {
        perror(message);
    } else {
        printf("ERROR: %s\n", message);
    }

    exit(1);
}
```

```c
void Address_print(struct Address *addr)
{
    printf("%d %s %s\n",
            addr->id, addr->name, addr->email);
}

void Database_load(struct Connection *conn)
{
    int rc = fread(conn->db, sizeof(struct Database), 1, conn->f
ile);
    if(rc != 1) die("Failed to load database.");
}

struct Connection *Database_open(const char *filename, char mode)

{
    struct Connection *conn = malloc(sizeof(struct Connection));
    if(!conn) die("Memory error");

    conn->db = malloc(sizeof(struct Database));
    if(!conn->db) die("Memory error");

    if(mode == 'c') {
        conn->file = fopen(filename, "w");
    } else {
        conn->file = fopen(filename, "r+");

        if(conn->file) {
            Database_load(conn);
        }
    }

    if(!conn->file) die("Failed to open the file");

    return conn;
}

void Database_close(struct Connection *conn)
{
    if(conn) {
        if(conn->file) fclose(conn->file);
        if(conn->db) free(conn->db);
        free(conn);
    }
}

void Database_write(struct Connection *conn)
{
    rewind(conn->file);

    int rc = fwrite(conn->db, sizeof(struct Database), 1, conn->
file);
```

```c
    if(rc != 1) die("Failed to write database.");

    rc = fflush(conn->file);
    if(rc == -1) die("Cannot flush database.");
}

void Database_create(struct Connection *conn)
{
    int i = 0;

    for(i = 0; i < MAX_ROWS; i++) {
        // make a prototype to initialize it
        struct Address addr = {.id = i, .set = 0};
        // then just assign it
        conn->db->rows[i] = addr;
    }
}

void Database_set(struct Connection *conn, int id, const char *n
ame, const char *email)
{
    struct Address *addr = &conn->db->rows[id];
    if(addr->set) die("Already set, delete it first");

    addr->set = 1;
    // WARNING: bug, read the "How To Break It" and fix this
    char *res = strncpy(addr->name, name, MAX_DATA);
    // demonstrate the strncpy bug
    if(!res) die("Name copy failed");

    res = strncpy(addr->email, email, MAX_DATA);
    if(!res) die("Email copy failed");
}

void Database_get(struct Connection *conn, int id)
{
    struct Address *addr = &conn->db->rows[id];

    if(addr->set) {
        Address_print(addr);
    } else {
        die("ID is not set");
    }
}

void Database_delete(struct Connection *conn, int id)
{
    struct Address addr = {.id = id, .set = 0};
    conn->db->rows[id] = addr;
}

void Database_list(struct Connection *conn)
{
```

```
    int i = 0;
    struct Database *db = conn->db;

    for(i = 0; i < MAX_ROWS; i++) {
        struct Address *cur = &db->rows[i];

        if(cur->set) {
            Address_print(cur);
        }
    }
}

int main(int argc, char *argv[])
{
    if(argc < 3) die("USAGE: ex17 <dbfile> <action> [action para
ms]");

    char *filename = argv[1];
    char action = argv[2][0];
    struct Connection *conn = Database_open(filename, action);
    int id = 0;

    if(argc > 3) id = atoi(argv[3]);
    if(id >= MAX_ROWS) die("There's not that many records.");

    switch(action) {
        case 'c':
            Database_create(conn);
            Database_write(conn);
            break;

        case 'g':
            if(argc != 4) die("Need an id to get");

            Database_get(conn, id);
            break;

        case 's':
            if(argc != 6) die("Need id, name, email to set");

            Database_set(conn, id, argv[4], argv[5]);
            Database_write(conn);
            break;

        case 'd':
            if(argc != 4) die("Need id to delete");

            Database_delete(conn, id);
            Database_write(conn);
            break;

        case 'l':
            Database_list(conn);
```

```
                break;
            default:
                die("Invalid action, only: c=create, g=get, s=set, d
=del, l=list");
    }

    Database_close(conn);

    return 0;
}
```

在这个程序中我使用了一系列的结构来创建用于地址薄的小型数据库。其中，我是用了一些你从来没见过的东西，所以你应该逐行浏览这段代码，解释每一行做了什么，并且查询你不认识的任何函数。下面是你需要注意的几个关键部分：

`#define` 常量

我使用了“C预处理器”的另外一部分，来创建 `MAX_DATA` 和 `MAX_ROWS` 的设置常量。我之后会更多地讲解预处理器的功能，不过这是一个创建可靠的常量的简易方法。除此之外还有另一种方法，但是在特定场景下并不适用。

定长结构体

`Address` 结构体接着使用这些常量来创建数据，这些数据是定长的，它们并不高效，但是便于存储和读取。 `Database` 结构体也是定长的，因为它有一个定长的 `Address` 结构体数组。这样你就可以稍后把整个数据一步写到磁盘。

出现错误时终止的 `die` 函数

在像这样的小型程序中，你可以编写一个单个函数在出现错误时杀掉程序。我把它叫做 `die` 。而且在任何失败的函数调用，或错误输出之后，它会调用 `exit` 带着错误退出程序。

用于错误报告的 `errno` 和 `perror`

当函数返回了一个错误时，它通常设置一个叫做 `errno` 的“外部”变量，来描述发生了什么错误。它们只是数字，所以你可以使用 `perror` 来“打印出错误信息”。

文件函数

我使用了一些新的函数，比如 `fopen` ， `fread` ， `fclose` ，和 `rewind` 来处理文件。这些函数中每个都作用于 `FILE` 结构体上，就像你的结构体似的，但是它由C标准库定义。

嵌套结构体指针

你应该学习这里的嵌套结构器和获取数组元素地址的用法，它读作“读取 `db` 中的 `conn` 中的 `rows` 的第 `i` 个元素，并返回地址（ `&` ）”。

> 译者注：这里有个更简便的写法是 `db->conn->row + i` 。

结构体原型的复制

它在 `Database_delete` 中体现得最清楚，你可以看到我是用了临时的局部 `Address` 变量，初始化了它的 `id` 和 `set` 字段，接着通过把它赋值给 `rows` 数组中的元素，简单地复制到数组中。这个小技巧确保了所有除了 `set` 和 `id` 的字段都初始化为0，而且很容易编写。顺便说一句，你不应该在这种数组复制操作中使用 `memcpy` 。现代C语言中你可以只是将一个赋值给另一个，它会自动帮你处理复制。

处理复杂参数

我执行了一些更复杂的参数解析，但是这不是处理它们的最好方法。在这本书的后面我们将会了解一些用于解析的更好方法。

将字符串转换为整数

我使用了 `atoi` 函数在命令行中接受作为id的字符串并把它转换为 `int id` 变量。去查询这个函数以及相似的函数。

在堆上分配大块数据

这个程序的要点就是在我创建 `Database` 的时候，我使用了 `malloc` 来向OS请求一块大容量的内存。稍后我会讲得更细致一些。

`NULL` 就是0，所以可转成布尔值

在许多检查中，我简单地通过 `if(!ptr) die("fail!")` 检测了一个指针是不是 `NULL` 。这是有效的，因为 `NULL` 会被计算成假。在一些少见的系统中， `NULL` 会储存在计算机中，并且表示为一些不是0的东西。但在C标准中，你可以把它当成0来编写代码。到目前为止，当我说“ `NULL` 就是0”的时候，我都是对一些迂腐的人说的。

你会看到什么

你应该为此花费大量时间，知道你可以测试它能正常工作了。并且你应当用 `Valgrind` 来确保你在所有地方都正确使用内存。下面是我的测试记录，并且随后使用了 `Valgrind` 来检查操作：

```
$ make ex17
cc -Wall -g    ex17.c   -o ex17
$ ./ex17 db.dat c
$ ./ex17 db.dat s 1 zed zed@zedshaw.com
$ ./ex17 db.dat s 2 frank frank@zedshaw.com
$ ./ex17 db.dat s 3 joe joe@zedshaw.com
$
$ ./ex17 db.dat l
1 zed zed@zedshaw.com
2 frank frank@zedshaw.com
3 joe joe@zedshaw.com
$ ./ex17 db.dat d 3
$ ./ex17 db.dat l
1 zed zed@zedshaw.com
2 frank frank@zedshaw.com
$ ./ex17 db.dat g 2
2 frank frank@zedshaw.com
$
$ valgrind --leak-check=yes ./ex17 db.dat g 2
# cut valgrind output...
$
```

`Valgrind` 实际的输出没有显式，因为你应该能够发现它。

> 注
>
> `Vagrind` 可以报告出你泄露的小块内存，但是它有时会过度报告OSX内部的API。如果你发现它显示了不属于你代码中的泄露，可以忽略它们。

堆和栈的内存分配

对于现在你们这些年轻人来说，编程简直太容易了。如果你玩玩Ruby或者Python的话，只要创建对象或变量就好了，不用管它们存放在哪里。你并不关心它们是否存放在栈上或堆上。你的编程语言甚至完全不会把变量放在栈上，它们都在堆上，并且你也不知道是否是这样。

然而C完全不一样，因为它使用了CPU真实的机制来完成工作，这涉及到RAM中的一块叫做栈的区域，以及另外一块叫做堆的区域。它们的差异取决于取得储存空间的位置。

堆更容易解释，因为它就是你电脑中的剩余内存，你可以通过 `malloc` 访问它来获取更多内存，OS会使用内部函数为你注册一块内存区域，并且返回指向它的指针。当你使用完这片区域时，你应该使用 `free` 把它交还给OS，使之能被其它程序复用。如果你不这样做就会导致程序“泄露”内存，但是 `Valgrind` 会帮你监测这些内存泄露。

栈是一个特殊的内存区域，它储存了每个函数的创建的临时变量，它们对于该函数为局部变量。它的工作机制是，函数的每个函数都会“压入”栈中，并且可在函数内部使用。它是一个真正的栈数据结构，所以是后进先出的。这对于 `main` 中所有类似 `char section` 和 `int id` 的局部变量也是相同的。使用栈的优点是，当函数退出时C编译器会从栈中“弹出”所有变量来清理。这非常简单，也防止了栈上变量的内存泄露。

理清内存的最简单的方式是遵守这条原则：如果你的变量并不是从 `malloc` 中获取的，也不是从一个从 `malloc` 获取的函数中获取的，那么它在栈上。

下面是三个值得关注的关于栈和堆的主要问题：

- 如果你从 `malloc` 获取了一块内存，并且把指针放在了栈上，那么当函数退出时，指针会被弹出而丢失。
- 如果你在栈上存放了大量数据（比如大结构体和数组），那么会产生“栈溢出”并且程序会中止。这种情况下应该通过 `malloc` 放在堆上。
- 如果你获取了指向栈上变量的指针，并且将它用于传参或从函数返回，接收它的函数会产生“段错误”。因为实际的数据被弹出而消失，指针也会指向被释放的内存。

这就是我在程序中使用 `Database_open` 来分配内存或退出的原因，相应的 `Database_close` 用于释放内存。如果你创建了一个“创建”函数，它创建了一些东西，那么一个“销毁”函数可以安全地清理这些东西。这样会更容易理清内存。

最后，当一个程序退出时，OS会为你清理所有的资源，但是有时不会立即执行。一个惯用法（也是本次练习中用到的）是立即终止并且让OS清理错误。

如何使它崩溃

这个程序有很多可以使之崩溃的地方，尝试下面这些东西，同时也想出自己的办法。

- 最经典的方法是移除一些安全检查，你就可以传入任意数据。例如，第160行的检查防止你传入任何记录序号。
- 你也可以尝试弄乱数据文件。使用任何编辑器打开它并且随机修改几个字节并关闭。
- 你也可以寻找在运行中向程序传递非法参数的办法。例如将文件参数放到动作后面，就会创建一个以动作命名的文件，并且按照文件名的第一个字符执行动作。
- 这个程序中有个bug，因为 `strncpy` 有设计缺陷。查询 `strncpy` 的相关资料，然后试着弄清楚如果 `name` 或者 `address` 超过512个字节会发生什么。可以通过简单把最后一个字符设置成 `'\0'` 来修复它，你应该无论如何都这样做（这也是函数原本应该做的）。
- 在附加题中我会让你传递参数来创建任意大小的数据库。在你造成程序退出或 `malloc` 的内存不足之前，尝试找出最大的数据库尺寸是多少。

附加题

- `die` 函数需要接收 `conn` 变量作为参数，以便执行清理并关闭它。
- 修改代码，使其接收参数作为 `MAX_DATA` 和 `MAX_ROWS` ，将它们储存在 `Database` 结构体中，并且将它们写到文件。这样就可以创建任意大小的数据库。
- 向数据库添加更多操作，比如 `find` 。
- 查询C如何打包结构体，并且试着弄清楚为什么你的文件是相应的大小。看看你是否可以计算出结构体添加一些字段之后的新大小。
- 向 `Address` 添加一些字段，使它们可被搜索。
- 编写一个shell脚本来通过以正确顺序运行命令执行自动化测试。提示：在 `bash` 顶端使用使用 `set -e` ，使之在任何命令发生错误时退出。

  > 译者注：使用Python编写多行脚本或许更方便一些。

- 尝试重构程序，使用单一的全局变量来储存数据库连接。这个新版本和旧版本比起来如何？
- 搜索“栈数据结构”，并且在你最喜欢的语言中实现它，然后尝试在C中实现。

练习18：函数指针

> 原文：Exercise 18: Pointers To Functions
>
> 译者：飞龙

函数在C中实际上只是指向程序中某一个代码存在位置的指针。就像你创建过的结构体指针、字符串和数组那样，你也可以创建指向函数的指针。函数指针的主要用途是向其他函数传递“回调”，或者模拟类和对象。在这个练习中我们会创建一些回调，并且下一节我们会制作一个简单的对象系统。

函数指针的格式类似这样：

```
int (*POINTER_NAME)(int a, int b)
```

记住如何编写它的一个方法是：

- 编写一个普通的函数声明： `int callme(int a, int b)`
- 将函数用指针语法包装： `int (*callme)(int a, int b)`
- 将名称改成指针名称： `int (*compare_cb)(int a, int b)`

这个方法的关键是，当你完成这些之后，指针的变量名称为 `compare_cb` ，而你可以将它用作函数。这类似于指向数组的指针可以表示所指向的数组。指向函数的指针也可以用作表示所指向的函数，只不过是不同的名字。

```
int (*tester)(int a, int b) = sorted_order;
printf("TEST: %d is same as %d\n", tester(2, 3), sorted_order(2,
3));
```

即使是对于返回指针的函数指针，上述方法依然有效：

- 编写： `char *make_coolness(int awesome_levels)`
- 包装： `char *(*make_coolness)(int awesome_levels)`
- 重命名： `char *(*coolness_cb)(int awesome_levels)`

需要解决的下一个问题是使用函数指针向其它函数提供参数比较困难，比如当你打算向其它函数传递回调函数的时候。解决方法是使用 `typedef` ，它是C的一个关键字，可以给其它更复杂的类型起个新的名字。你需要记住的事情是，将 `typedef` 添加到相同的指针语法之前，然后你就可以将那个名字用作类型了。我使用下面的代码来演示这一特性：

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

```
#include <string.h>

/** Our old friend die from ex17. */
void die(const char *message)
{
    if(errno) {
        perror(message);
    } else {
        printf("ERROR: %s\n", message);
    }

    exit(1);
}

// a typedef creates a fake type, in this
// case for a function pointer
typedef int (*compare_cb)(int a, int b);

/**
 * A classic bubble sort function that uses the
 * compare_cb to do the sorting.
 */
int *bubble_sort(int *numbers, int count, compare_cb cmp)
{
    int temp = 0;
    int i = 0;
    int j = 0;
    int *target = malloc(count * sizeof(int));

    if(!target) die("Memory error.");

    memcpy(target, numbers, count * sizeof(int));

    for(i = 0; i < count; i++) {
        for(j = 0; j < count - 1; j++) {
            if(cmp(target[j], target[j+1]) > 0) {
                temp = target[j+1];
                target[j+1] = target[j];
                target[j] = temp;
            }
        }
    }

    return target;
}

int sorted_order(int a, int b)
{
    return a - b;
}

int reverse_order(int a, int b)
{
```

```
    return b - a;
}

int strange_order(int a, int b)
{
    if(a == 0 || b == 0) {
        return 0;
    } else {
        return a % b;
    }
}

/**
 * Used to test that we are sorting things correctly
 * by doing the sort and printing it out.
 */
void test_sorting(int *numbers, int count, compare_cb cmp)
{
    int i = 0;
    int *sorted = bubble_sort(numbers, count, cmp);

    if(!sorted) die("Failed to sort as requested.");

    for(i = 0; i < count; i++) {
        printf("%d ", sorted[i]);
    }
    printf("\n");

    free(sorted);
}


int main(int argc, char *argv[])
{
    if(argc < 2) die("USAGE: ex18 4 3 1 5 6");

    int count = argc - 1;
    int i = 0;
    char **inputs = argv + 1;

    int *numbers = malloc(count * sizeof(int));
    if(!numbers) die("Memory error.");

    for(i = 0; i < count; i++) {
        numbers[i] = atoi(inputs[i]);
    }

    test_sorting(numbers, count, sorted_order);
    test_sorting(numbers, count, reverse_order);
    test_sorting(numbers, count, strange_order);

    free(numbers);
```

```
    return 0;
}
```

在这段程序中，你将创建动态排序的算法，它会使用比较回调对整数数组排序。下面是这个程序的分解，你应该能够清晰地理解它。

ex18.c:1~6

通常的包含，用于所调用的所有函数。

ex18.c:7~17

这就是之前练习的 `die` 函数，我将它用于错误检查。

ex18.c:21

这是使用 `typedef` 的地方，在后面我像 `int` 或 `char` 类型那样，在 `bubble_sort` 和 `test_sorting` 中使用了 `compare_cb` 。

ex18.c:27~49

一个冒泡排序的实现，它是整数排序的一种不高效的方法。这个函数包含了：

ex18.c:27

这里是将 `typedef` 用于 `compare_cb` 作为 `cmp` 最后一个参数的地方。现在它是一个会返回两个整数比较结果用于排序的函数。

ex18.c:29~34

栈上变量的通常创建语句，前面是使用 `malloc` 创建的堆上整数数组。确保你理解了 `count * sizeof(int)` 做了什么。

ex18.c:38

冒泡排序的外循环。

ex18.c:39

冒泡排序的内循环。

ex18.c:40

现在我调用了 `cmp` 回调，就像一个普通函数那样，但是不通过预先定义好的函数名，而是一个指向它的指针。调用者可以像它传递任何参数，只要这些参数符合 `compare_cb` `typedef` 的签名。

ex18.c:41-43

冒泡排序所需的实际交换操作。

ex18.c:48

最后返回新创建和排序过的结果数据 `target` 。

ex18.c:51-68

`compare_cb`函数类型三个不同版本，它们需要和我们所创建的`typedef`具有相同的定义。否则C编辑器会报错说类型不匹配。

ex18.c:74-87

这是`bubble_sort`函数的测试。你可以看到我同时将`compare_cb`传给了`bubble_sort`来演示它是如何像其它指针一样传递的。

ex18.c:90-103

一个简单的主函数，基于你通过命令行传递进来的整数，创建了一个数组。然后调用了`test_sorting`函数。

ex18.c:105-107

最后，你会看到`compare_cb`函数指针的`typedef`是如何使用的。我仅仅传递了`sorted_order`、`reverse_order`和`strange_order`的名字作为函数来调用`test_sorting`。C编译器会找到这些函数的地址，并且生成指针用于`test_sorting`。如果你看一眼`test_sorting`你会发现它把这些函数传给了`bubble_sort`，并不关心它们是做了什么。只要符合`compare_cb`原型的东西都有效。

ex18.c:109

我们在最后释放了我们创建的整数数组。

你会看到什么

运行这个程序非常简单，但是你要尝试不同的数字组合，甚至要尝试输入非数字来看看它做了什么：

```
$ make ex18
cc -Wall -g    ex18.c   -o ex18
$ ./ex18 4 1 7 3 2 0 8
0 1 2 3 4 7 8
8 7 4 3 2 1 0
3 4 2 7 1 0 8
$
```

如何使它崩溃

我打算让你做一些奇怪的事情来使它崩溃，这些函数指针都是类似于其它指针的指针，他们都指向内存的一块区域。C中可以将一种指针的指针转换为另一种，以便以不同方式处理数据。这些通常是不必要的，但是为了想你展示如何侵入你的电脑，我希望你把这段代码添加在`test_sorting`下面：

```c
unsigned char *data = (unsigned char *)cmp;

for(i = 0; i < 25; i++) {
    printf("%02x:", data[i]);
}

printf("\n");
```

这个循环将你的函数转换成字符串，并且打印出来它的内容。这并不会中断你的程序，除非CPU和OS在执行过程中遇到了问题。在它打印排序过的数组之后，你所看到的是一个十六进制数字的字符串：

```
55:48:89:e5:89:7d:fc:89:75:f8:8b:55:fc:8b:45:f8:29:d0:c9:c3:55:4
8:89:e5:89:
```

这就应该是函数的原始的汇编字节码了，你应该能看到它们有相同的起始和不同的结尾。也有可能这个循环并没有获得函数的全部，或者获得了过多的代码而跑到程序的另外一片空间。这些不通过更多分析是不可能知道的。

附加题

- 用十六进制编辑器打开 `ex18` ，接着找到函数起始处的十六进制代码序列，看看是否能在原始程序中找到函数。
- 在你的十六进制编辑器中找到更多随机出现的东西并修改它们。重新运行你的程序看看发生了什么。字符串是你最容易修改的东西。
- 将错误的函数传给 `compare_cb` ，并看看C编辑器会报告什么错误。
- 将 `NULL` 传给它，看看程序中会发生什么。然后运行 `Valgrind` 来看看它会报告什么。
- 编写另一个排序算法，修改 `test_sorting` 使它接收任意的排序函数和排序函数的比较回调。并使用它来测试两种排序算法。

练习19：一个简单的对象系统

> 原文：Exercise 19: A Simple Object System
>
> 译者：飞龙

我在学习面向对象编程之前学了C，所以它有助于我在C中构建面向对象系统，来理解OOP的基本含义。你可能在学习C之前就学了OOP语言，所以这章也可能会起到一种衔接作用。这个联系中，你将会构建一个简单的对象系统，但是也会了解更多关于C预处理器的事情。

这个练习会构建一个简单的游戏，在游戏中你会在一个小型的城堡中杀死弥诺陶洛斯，并没有任何神奇之处，只是四个房间和一个坏家伙。这个练习同时是一个多文件的项目，并且比起之前的一些程序看起来更像一个真正的C程序。我在这里介绍C预处理器的原因，是你需要它来在你自己的程序中创建多个文件。

C预处理器如何工作

C预处理器是个模板处理系统，它主要的用途是让C代码的编程更加容易，但是它通过一个语法感知的模板机制来实现。以前人们主要使用C预处理器来储存常量，以及创建“宏”来简化复杂的代码。在现代C语言中你会实际上使用它作为代码生成器来创建模板化的代码片段。

C预处理器的工作原理是，如果你给它一个文件，比如 `.c` 文件，它会处理以 `#` （井号）字符开头的各种文本。当它遇到一个这样的文本时，它会对输入文件中的文本做特定的替换。C预处理器的主要优点是他可以包含其他文件，并且基于该文件的内容对它的宏列表进行扩展。

一个快速查看预处理器所做事情的方法，是对上个练习中的代码执行下列命令：

```
cpp ex18.c | less
```

这会产生大量输出，但是如果你滚动它，会看到你使用 `#include` 包含的其他文件的内容。在原始的代码中向下滚动，你可以看到 `cpp` 如何基于头文件中不同的 `#define` 宏来转换代码。

C编译器与 `cpp` 的集成十分紧密，这个例子只是向你展示它是如何在背后工作的。在现代C语言中， `cpp` 系统也集成到C的函数中，你或许可以将它当做C语言的一部分。

在剩余的章节中，我们会使用更多预处理器的语法，并且像往常一样解释它们。

原型对象系统

我们所创建的OOP系统是一个简单的“原型”风格的对象系统，很像JavaScript。你将以设置为字段的原型来开始，而不是类，接着将他们用作创建其它对象实例的基础。这个“没有类”的设计比起传统的基于类的对象系统更加易于实现和使用。

Object头文件

我打算将数据类型和函数声明放在一个单独的头文件中，叫做 `object.h` 。这个是一个标准的C技巧，可以让你集成二进制库，但其它程序员任然需要编译。在这个文件中，我使用了多个高级的C预处理器技巧，我接下来准备简略地描述它们，并且你会在后续的步骤中看到。

```
#ifndef _object_h
#define _object_h

typedef enum {
    NORTH, SOUTH, EAST, WEST
} Direction;

typedef struct {
    char *description;
    int (*init)(void *self);
    void (*describe)(void *self);
    void (*destroy)(void *self);
    void *(*move)(void *self, Direction direction);
    int (*attack)(void *self, int damage);
} Object;

int Object_init(void *self);
void Object_destroy(void *self);
void Object_describe(void *self);
void *Object_move(void *self, Direction direction);
int Object_attack(void *self, int damage);
void *Object_new(size_t size, Object proto, char *description);

#define NEW(T, N) Object_new(sizeof(T), T##Proto, N)
#define _(N) proto.N

#endif
```

看一看这个文件，你会发现我使用了几个新的语法片段，你之前从来没见过它们：

`#ifndef`

你已经见过了用于创建简单常量的 `#define` ，但是C预处理器可以根据条件判断来忽略一部分代码。这里的 `#ifndef` 是“如果没有被定义”的意思，它会检查是否已经出现过 `#define _object_h` ，如果已出现，就跳过这段代码。我之所以这样写，是因为我们可以将这个文件包含任意次，而无需担心多次定义里面的东西。

`#define`

有了上面保护该文件的 `#ifndef` ，我们接着添加 `_object_h` 的定义，因此之后任何试图包含此文件的行为，都会由于上面的语句而跳过这段代码。

`#define NEW(T,N)`

这条语句创建了一个宏，就像模板函数一样，无论你在哪里编写左边的代码，都会展开成右边的代码。这条语句仅仅是对我们通常调用的 `Object_new` 制作了一个快捷方式，并且避免了潜在的调用错误。在宏这种工作方式下， `T` 、 `N` 还有 `New` 都被“注入”进了右边的代码中。 `T##Proto` 语法表示“将Proto连接到T的末尾”，所以如果你写下 `NEW(Room, "Hello.")` ，就会在这里变成 `RoomProto` 。

`#define _(N)`

这个宏是一种为对象系统设计的“语法糖”，将 `obj->proto.blah` 简写为 `obj->_(blah)` 。它不是必需的，但是它是一个接下来会用到的有趣的小技巧。

Object源文件

`object.h` 是声明函数和数据类型的地方，它们在 `object.c` 中被定义（创建），所以接下来：

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "object.h"
#include <assert.h>

void Object_destroy(void *self)
{
    Object *obj = self;

    if(obj) {
        if(obj->description) free(obj->description);
        free(obj);
    }
}

void Object_describe(void *self)
{
    Object *obj = self;
    printf("%s.\n", obj->description);
}

int Object_init(void *self)
{
    // do nothing really
    return 1;
}
```

```c
void *Object_move(void *self, Direction direction)
{
    printf("You can't go that direction.\n");
    return NULL;
}

int Object_attack(void *self, int damage)
{
    printf("You can't attack that.\n");
    return 0;
}

void *Object_new(size_t size, Object proto, char *description)
{
    // setup the default functions in case they aren't set
    if(!proto.init) proto.init = Object_init;
    if(!proto.describe) proto.describe = Object_describe;
    if(!proto.destroy) proto.destroy = Object_destroy;
    if(!proto.attack) proto.attack = Object_attack;
    if(!proto.move) proto.move = Object_move;

    // this seems weird, but we can make a struct of one size,
    // then point a different pointer at it to "cast" it
    Object *el = calloc(1, size);
    *el = proto;

    // copy the description over
    el->description = strdup(description);

    // initialize it with whatever init we were given
    if(!el->init(el)) {
        // looks like it didn't initialize properly
        el->destroy(el);
        return NULL;
    } else {
        // all done, we made an object of any type
        return el;
    }
}
```

这个文件中并没有什么新东西，除了一个小技巧之外。 `Object_new` 函数通过把原型放到结构体的开头，利用了 `structs` 工作机制的一个方面。当你在之后看到 `ex19.h` 头文件时，你会明白为什么我将 `Object` 作为结构体的第一个字段。由于C按顺序将字段放入结构体，并且由于指针可以指向一块内存，我就可以将指针转换为任何我想要的东西。在这种情况下，即使我通过 `calloc` 获取了一大块内存，我仍然可以使用 `Object` 指针来指向它。

当我开始编写 `ex19.h` 文件时，我会把它解释得更详细一些，因为当你看到它怎么用的时候才能更容易去理解它。

上面的代码创建了基本的对象系统，但是你需要编译它和将它链接到`ex19.c`文件，来创建出完整的程序。`object.c`文件本身并没有`main`函数，所以它不可能被编译为完整的程序。下面是一个`Makefile`文件，它基于已经完成的事情来构建程序：

```
CFLAGS=-Wall -g

all: ex19

ex19: object.o

clean:
  rm -f ex19
```

这个`Makefile`所做的事情仅仅是让`ex19`依赖于`object.o`。还记得`make`可以根据扩展名构建不同的文件吗？这相当于告诉`make`执行下列事情：

- 当我运行`make`时，默认的`all`会构建`ex19`。
- 当它构建`ex19`时，也需要构建`object.o`，并且将它包含在其中。
- `make`并不能找到`object.o`，但是它能发现`object.c`文件，并且知道如何把`.c`文件变成`.o`文件，所以它就这么做了。
- 一旦`object.o`文件构建完成，它就会运行正确的编译命令，从`ex19.c`和`object.o`中构建`ex19`。

游戏实现

一旦你编写完成了那些文件，你需要使用对象系统来实现实际的游戏，第一步就是把所有数据类型和函数声明放在`ex19.h`文件中：

```c
#ifndef _ex19_h
#define _ex19_h

#include "object.h"

struct Monster {
    Object proto;
    int hit_points;
};

typedef struct Monster Monster;

int Monster_attack(void *self, int damage);
int Monster_init(void *self);

struct Room {
    Object proto;

    Monster *bad_guy;

    struct Room *north;
    struct Room *south;
    struct Room *east;
    struct Room *west;
};

typedef struct Room Room;

void *Room_move(void *self, Direction direction);
int Room_attack(void *self, int damage);
int Room_init(void *self);


struct Map {
    Object proto;
    Room *start;
    Room *location;
};

typedef struct Map Map;

void *Map_move(void *self, Direction direction);
int Map_attack(void *self, int damage);
int Map_init(void *self);

#endif
```

它创建了三个你将会用到的新对象：`Monster`，`Room`，和`Map`。

看一眼 `object.c:52` ，你可以看到这是我使用 `Object *el = calloc(1, size)` 的地方。回去看 `object.h` 的 `NEW` 宏，你可以发现它获得了另一个结构体的 `sizeof` ，比如 `Room` ，并且分配了这么多的空间。然而，由于我像一个 `Object` 指针指向了这块内存，并且我在 `Room` 的开头放置了 `Object proto` ，所以就可以将 `Room` 当成 `Object` 来用。

详细分解请见下面：

- 我调用了 `NEW(Room, "Hello.")` ，C预处理器会将其展开为 `Object_new(sizeof(Room), RoomProto, "Hello.")` 。
- 执行过程中，在 `Object_new` 的内部我分配了 `Room` 大小的一块内存，但是用 `Object *el` 来指向它。
- 由于C将 `Room.proto` 字段放在开头，这意味着 `el` 指针实际上指向了能访问到完整 `Object` 结构体的，足够大小的一块内存。它不知道这块内存叫做 `proto` 。
- 接下来它使用 `Object *el` 指针，通过 `*el = proto` 来设置这块内存的内容。要记住你可以复制结构体，而且 `*el` 的意思是“ `el` 所指向对象的值”，所以整条语句意思是“将 `el` 所指向对象的值赋为 `proto` ”。
- 由于这个谜之结构体被填充为来自 `proto` 的正确数据，这个函数接下来可以在 `Object` 上调用 `init` ，或者 `destroy` 。但是最神奇的一部分是无论谁调用这个函数都可以将它们改为想要的东西。

结合上面这些东西，我就可以使用这一个函数来创建新的类型，并且向它们提供新的函数来修改它们的行为。这看起来像是“黑魔法”，但它是完全有效的C代码。实际上，有少数标准的系统函数也以这种方式工作，我们将会用到一些这样的函数在网络程序中转换地址。

编写完函数定义和数据结构之后，我现在就可以实现带有四个房间和一个牛头人的游戏了。

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "ex19.h"


int Monster_attack(void *self, int damage)
{
    Monster *monster = self;

    printf("You attack %s!\n", monster->_(description));

    monster->hit_points -= damage;

    if(monster->hit_points > 0) {
        printf("It is still alive.\n");
        return 0;
    } else {
```

```
            printf("It is dead!\n");
            return 1;
        }
}

int Monster_init(void *self)
{
    Monster *monster = self;
    monster->hit_points = 10;
    return 1;
}

Object MonsterProto = {
    .init = Monster_init,
    .attack = Monster_attack
};


void *Room_move(void *self, Direction direction)
{
    Room *room = self;
    Room *next = NULL;

    if(direction == NORTH && room->north) {
        printf("You go north, into:\n");
        next = room->north;
    } else if(direction == SOUTH && room->south) {
        printf("You go south, into:\n");
        next = room->south;
    } else if(direction == EAST && room->east) {
        printf("You go east, into:\n");
        next = room->east;
    } else if(direction == WEST && room->west) {
        printf("You go west, into:\n");
        next = room->west;
    } else {
        printf("You can't go that direction.");
        next = NULL;
    }

    if(next) {
        next->_(describe)(next);
    }

    return next;
}


int Room_attack(void *self, int damage)
{
    Room *room = self;
    Monster *monster = room->bad_guy;
```

```
    if(monster) {
        monster->_(attack)(monster, damage);
        return 1;
    } else {
        printf("You flail in the air at nothing. Idiot.\n");
        return 0;
    }
}


Object RoomProto = {
    .move = Room_move,
    .attack = Room_attack
};


void *Map_move(void *self, Direction direction)
{
    Map *map = self;
    Room *location = map->location;
    Room *next = NULL;

    next = location->_(move)(location, direction);

    if(next) {
        map->location = next;
    }

    return next;
}

int Map_attack(void *self, int damage)
{
    Map* map = self;
    Room *location = map->location;

    return location->_(attack)(location, damage);
}


int Map_init(void *self)
{
    Map *map = self;

    // make some rooms for a small map
    Room *hall = NEW(Room, "The great Hall");
    Room *throne = NEW(Room, "The throne room");
    Room *arena = NEW(Room, "The arena, with the minotaur");
    Room *kitchen = NEW(Room, "Kitchen, you have the knife now")
;

    // put the bad guy in the arena
    arena->bad_guy = NEW(Monster, "The evil minotaur");
```

```c
    // setup the map rooms
    hall->north = throne;

    throne->west = arena;
    throne->east = kitchen;
    throne->south = hall;

    arena->east = throne;
    kitchen->west = throne;

    // start the map and the character off in the hall
    map->start = hall;
    map->location = hall;

    return 1;
}

Object MapProto = {
    .init = Map_init,
    .move = Map_move,
    .attack = Map_attack
};

int process_input(Map *game)
{
    printf("\n> ");

    char ch = getchar();
    getchar(); // eat ENTER

    int damage = rand() % 4;

    switch(ch) {
        case -1:
            printf("Giving up? You suck.\n");
            return 0;
            break;

        case 'n':
            game->_(move)(game, NORTH);
            break;

        case 's':
            game->_(move)(game, SOUTH);
            break;

        case 'e':
            game->_(move)(game, EAST);
            break;

        case 'w':
            game->_(move)(game, WEST);
```

```
                break;

            case 'a':

                game->_(attack)(game, damage);
                break;
            case 'l':
                printf("You can go:\n");
                if(game->location->north) printf("NORTH\n");
                if(game->location->south) printf("SOUTH\n");
                if(game->location->east) printf("EAST\n");
                if(game->location->west) printf("WEST\n");
                break;

            default:
                printf("What?: %d\n", ch);
        }

        return 1;
}

int main(int argc, char *argv[])
{
    // simple way to setup the randomness
    srand(time(NULL));

    // make our map to work with
    Map *game = NEW(Map, "The Hall of the Minotaur.");

    printf("You enter the ");
    game->location->_(describe)(game->location);

    while(process_input(game)) {
    }

    return 0;
}
```

说实话这里面并没有很多你没有见过的东西，并且你只需要理解我使用头文件中宏的方法。下面是需要学习和理解的一些重要的核心知识：

- 实现一个原型涉及到创建它的函数版本，以及随后创建一个以“Proto”结尾的单一结构体。请参照 `MonsterProto` ， `RoomProto` 和 `MapProto` 。
- 由于 `Object_new` 的实现方式，如果你没有在你的原型中设置一个函数，它会获得在 `object.c` 中创建的默认实现。
- 在 `Map_init` 中我创建了一个微型世界，然而更重要的是我使用了 `object.h` 中的 `NEW` 宏来创建全部对象。要把这一概念记在脑子里，可以试着把使用 `NEW` 的地方替换成 `Object_new` 的直接调用，来观察它如何被替换。
- 使用这些对象涉及到在它们上面调用函数， `_(N)` 为我做了这些事情。如果你

观察代码 `monster->_(attack)(monster, damage)` ，你会看到我使用了宏将其替换成 `monster->proto.attack(monster, damage)` 。通过重新将这些调用写成原始形式来再次学习这个转换。另外，如果你被卡住了，手动运行 `cpp` 来查看究竟发生了什么。
- 我使用了两个新的函数 `srand` 和 `rand` ，它们可以设置一个简单的随机数生成器，对于游戏已经够用了。我也使用了 `time` 来初始化随机数生成器。试着研究它们。
- 我使用了一个新的函数 `getchar` 来从标准输入中读取单个字符。试着研究它。

你会看到什么

下面是我自己的游戏的输出：

```
$ make ex19
cc -Wall -g   -c -o object.o object.c
cc -Wall -g    ex19.c object.o   -o ex19
$ ./ex19
You enter the The great Hall.

> l
You can go:
NORTH

> n
You go north, into:
The throne room.

> l
You can go:
SOUTH
EAST
WEST

> e
You go east, into:
Kitchen, you have the knife now.

> w
You go west, into:
The throne room.

> s
You go south, into:
The great Hall.

> n
You go north, into:
The throne room.

> w
You go west, into:
The arena, with the minotaur.

> a
You attack The evil minotaur!
It is still alive.

> a
You attack The evil minotaur!
It is dead!

> ^D
Giving up? You suck.
$
```

审计该游戏

我把所有`assert`检查留给你作为练习，我通常把它们作为软件的一部分。你已经看到了我如何使用`assert`来保证程序正确运行。然而现在我希望你返回去并完成下列事情：

- 查看你定义的每个函数，一次一个文件。
- 在每个函数的最上面，添加`assert`来保证参数正确。例如在`Object_new`中要添加`assert(description != NULL)`。
- 浏览函数的每一行，找到所调用的任何函数。阅读它们的文档（或手册页），确认它们在错误下返回什么。添加另一个断言来检查错误是否发生。例如，`Object_new`在调用`calloc`之后应该进行`assert(el != NULL)`的检查。
- 如果函数应该返回一个值，也确保它返回了一个错误值（比如`NULL`），或者添加一个断言来确保返回值是有效的。例如，`Object_new`中，你需要在最后的返回之前添加`assert(el != NULL)`，由于它不应该为`NULL`。
- 对于每个你编写的`if`语句，确保都有对应的`else`语句，除非它用于错误检查并退出。
- 对于每个你编写的`switch`语句，确保都有一个`default`分支，来处理非预期的任何情况。

花费一些时间浏览函数的每一行，并且找到你犯下的任何错误。记住这个练习的要点是从“码农”转变为“黑客”。试着找到使它崩溃的办法，然后尽可能编写代码来防止崩溃或者过早退出。

附加题

- 修改`Makefile`文件，使之在执行`make clean`时能够同时清理`object.o`。
- 编写一个测试脚本，能够以多种方式来调用该游戏，并且扩展`Makefile`使之能够通过运行`make test`来测试该游戏。
- 在游戏中添加更多房间和怪物。
- 把游戏的逻辑放在其它文件中，并把它编译为`.o`。然后，使用它来编写另一个小游戏。如果你正确编写的话，你会在新游戏中创建新的`Map`和`main`函数。

练习20：Zed的强大的调试宏

> 原文：Exercise 20: Zed's Awesome Debug Macros
>
> 译者：飞龙

在C中有一个永恒的问题，它伴随了你很长时间，然而在这个练习我打算使用一系列我开发的宏来解决它。到现在为止你都不知道它们的强大之处，所以你必须使用它们，总有一天你会来找我说，“Zed，这些调试宏眞是太伟大了，我应该把我的第一个孩子的出生归功于你，因为你治好了我十年的心脏病，并且打消了我数次想要自杀的念头。眞是要谢谢你这样一个好人，这里有一百万美元，和Leo Fender设计的Snakehead Telecaster电吉他的原型。”

是的，它们的确很强大。

C的错误处理问题

几乎每个编程语言中，错误处理都非常难。有些语言尽可能试图避免错误这个概念，而另一些语言发明了复杂了控制结构，比如异常来传递错误状态。当然的错误大多是因为程序员假定错误不会发生，并且这一乐观的思想影响了他们所用和所创造的语言。

C通过返回错误码或设置全局的`errno`值来解决这些问题，并且你需要检查这些值。这种机制可以检查现存的复杂代码中，你执行的东西是否发生错误。当你编写更多的C代码时，你应该按照下列模式：

- 调用函数。
- 如果返回值出现错误（每次都必须检查）。
- 清理创建的所有资源。
- 打印出所有可能有帮助的错误信息。

这意味着对于每一个函数调用（是的，每个函数）你都可能需要多编写3~4行代码来确保它正常功能。这些还不包括清理你到目前创建的所有垃圾。如果你有10个不同的结构体，3个方式。和一个数据库链接，当你发现错误时你应该写额外的14行。

之前这并不是个问题，因为发生错误时，C程序会像你以前做的那样直接退出。你不需要清理任何东西，因为OS会为你自动去做。然而现在很多C程序需要持续运行数周、数月或者数年，并且需要优雅地处理来自于多种资源的错误。你并不能仅仅让你的服务器在首次运行就退出，你也不能让你写的库使使用它的程序退出。这非常糟糕。

其它语言通过异常来解决这个问题，但是这些问题也会在C中出现（其它语言也一样）。在C中你只能够返回一个值，但是异常是基于栈的返回系统，可以返回任意值。C语言中，尝试在栈上模拟异常非常困难，并且其它库也不会兼容。

调试宏

我使用的解决方案是，使用一系列“调试宏”，它们在C中实现了基本的调试和错误处理系统。这个系统非常易于理解，兼容于每个库，并且使C代码更加健壮和简洁。

它通过实现一系列转换来处理错误，任何时候发生了错误，你的函数都会跳到执行清理和返回错误代码的“error:”区域。你可以使用 `check` 宏来检查错误代码，打印错误信息，然后跳到清理区域。你也可以使用一系列日志函数来打印出有用的调试信息。

我现在会向你展示你目前所见过的，最强大且卓越的代码的全部内容。

```
#ifndef __dbg_h__
#define __dbg_h__

#include <stdio.h>
#include <errno.h>
#include <string.h>

#ifdef NDEBUG
#define debug(M, ...)
#else
#define debug(M, ...) fprintf(stderr, "DEBUG %s:%d: " M "\n", __
FILE__, __LINE__, ##__VA_ARGS__)
#endif

#define clean_errno() (errno == 0 ? "None" : strerror(errno))

#define log_err(M, ...) fprintf(stderr, "[ERROR] (%s:%d: errno:
%s) " M "\n", __FILE__, __LINE__, clean_errno(), ##__VA_ARGS__)

#define log_warn(M, ...) fprintf(stderr, "[WARN] (%s:%d: errno:
%s) " M "\n", __FILE__, __LINE__, clean_errno(), ##__VA_ARGS__)

#define log_info(M, ...) fprintf(stderr, "[INFO] (%s:%d) " M "\n"
, __FILE__, __LINE__, ##__VA_ARGS__)

#define check(A, M, ...) if(!(A)) { log_err(M, ##__VA_ARGS__); e
rrno=0; goto error; }

#define sentinel(M, ...)  { log_err(M, ##__VA_ARGS__); errno=0;
goto error; }

#define check_mem(A) check((A), "Out of memory.")

#define check_debug(A, M, ...) if(!(A)) { debug(M, ##__VA_ARGS__
); errno=0; goto error; }

#endif
```

是的，这就是全部代码了，下面是它每一行所做的事情。

dbg.h:1-2

防止意外包含多次的保护措施，你已经在上一个练习中见过了。

dbg.h:4-6

包含这些宏所需的函数。

dbg.h:8

`#ifdef` 的起始，它可以让你重新编译程序来移除所有调试日志信息。

dbg.h:9

如果你定义了 `NDEBUG` 之后编译，没有任何调试信息会输出。你可以看到 `#define debug()` 被替换为空（右边没有任何东西）。

dbg.h:10

上面的 `#ifdef` 所匹配的 `#else` 。

dbg.h:11

用于替代的 `#define debug` ，它将任何使用 `debug("format", arg1, arg2)` 的地方替换成 `fprintf` 对 `stderr` 的调用。许多程序员并不知道，但是你的确可以创建与 `printf` 类似的可变参数宏。许多C编译器（实际上是C预处理器）并不支持它，但是gcc可以做到。这里的魔法是使用 `##__VA_ARGS__` ，意思是将剩余的所有额外参数放到这里。同时也要注意，使用了 `__FILE__` 和 `__LINE__` 来获取当前 `fine:line` 用于调试信息。这会非常有帮助。

dbg.h:12

`#ifdef` 的结尾。

dbg.h:14

`clean_errno` 宏用于获取 `errno` 的安全可读的版本。中间奇怪的语法是“三元运算符”，你会在后面学到它。

dbg.h:16-20

`log_err` ， `log_warn` 和 `log_info` 宏用于为最终用户记录信息。它们类似于 `debug` 但不能被编译。

dbg.h:22

到目前为止最棒的宏。 `check` 会保证条件 `A` 为真，否则会记录错误 `M` （带着 `log_err` 的可变参数），之后跳到函数的 `error:` 区域来执行清理。

dbg.h:24

第二个最棒的宏，`sentinel`可以放在函数的任何不应该执行的地方，它会打印错误信息并且跳到`error:`标签。你可以将它放到`if-statements`或者`switch-statements`的不该被执行的分支中，比如`default`。

dbg.h:26

简写的`check_mem`宏，用于确保指针有效，否则会报告“内存耗尽”的错误。

dbg.h:28

用于替代的`check_debug`宏，它仍然会检查并处理错误，尤其是你并不想报告的普遍错误。它里面使用了`debug`代替`log_err`来报告错误，所以当你定义了`NDEBUG`，它仍然会检查并且发生错误时跳出，但是不会打印消息了。

使用dbg.h

下面是一个例子，在一个小的程序中使用了`dbg.h`的所有函数。这实际上并没有做什么事情，只是向你演示了如何使用每个宏。我们将在接下来的所有程序中使用这些宏，所有要确保理解了如何使用它们。

```c
#include "dbg.h"
#include <stdlib.h>
#include <stdio.h>


void test_debug()
{
    // notice you don't need the \n
    debug("I have Brown Hair.");

    // passing in arguments like printf
    debug("I am %d years old.", 37);
}

void test_log_err()
{
    log_err("I believe everything is broken.");
    log_err("There are %d problems in %s.", 0, "space");
}

void test_log_warn()
{
    log_warn("You can safely ignore this.");
    log_warn("Maybe consider looking at: %s.", "/etc/passwd");
}

void test_log_info()
{
    log_info("Well I did something mundane.");
    log_info("It happened %f times today.", 1.3f);
```

```
}

int test_check(char *file_name)
{
    FILE *input = NULL;
    char *block = NULL;

    block = malloc(100);
    check_mem(block); // should work

    input = fopen(file_name,"r");
    check(input, "Failed to open %s.", file_name);

    free(block);
    fclose(input);
    return 0;

error:
    if(block) free(block);
    if(input) fclose(input);
    return -1;
}

int test_sentinel(int code)
{
    char *temp = malloc(100);
    check_mem(temp);

    switch(code) {
        case 1:
            log_info("It worked.");
            break;
        default:
            sentinel("I shouldn't run.");
    }

    free(temp);
    return 0;

error:
    if(temp) free(temp);
    return -1;
}

int test_check_mem()
{
    char *test = NULL;
    check_mem(test);

    free(test);
    return 1;

error:
```

```
    return -1;
}

int test_check_debug()
{
    int i = 0;
    check_debug(i != 0, "Oops, I was 0.");

    return 0;
error:
    return -1;
}

int main(int argc, char *argv[])
{
    check(argc == 2, "Need an argument.");

    test_debug();
    test_log_err();
    test_log_warn();
    test_log_info();

    check(test_check("ex20.c") == 0, "failed with ex20.c");
    check(test_check(argv[1]) == -1, "failed with argv");
    check(test_sentinel(1) == 0, "test_sentinel failed.");
    check(test_sentinel(100) == -1, "test_sentinel failed.");
    check(test_check_mem() == -1, "test_check_mem failed.");
    check(test_check_debug() == -1, "test_check_debug failed.");

    return 0;

error:
    return 1;
}
```

要注意 `check` 是如何使用的，并且当它为 `false` 时会跳到 `error:` 标签来执行清理。这一行读作“检查A是否为真，不为真就打印M并跳出”。

你会看到什么

当你执行这段代码并且向第一个参数提供一些东西，你会看到：

```
$ make ex20
cc -Wall -g -DNDEBUG    ex20.c   -o ex20
$ ./ex20 test
[ERROR] (ex20.c:16: errno: None) I believe everything is broken.
[ERROR] (ex20.c:17: errno: None) There are 0 problems in space.
[WARN] (ex20.c:22: errno: None) You can safely ignore this.
[WARN] (ex20.c:23: errno: None) Maybe consider looking at: /etc/
passwd.
[INFO] (ex20.c:28) Well I did something mundane.
[INFO] (ex20.c:29) It happened 1.300000 times today.
[ERROR] (ex20.c:38: errno: No such file or directory) Failed to
open test.
[INFO] (ex20.c:57) It worked.
[ERROR] (ex20.c:60: errno: None) I shouldn't run.
[ERROR] (ex20.c:74: errno: None) Out of memory.
```

看到 `check` 失败之后，它是如何打印具体的行号了吗？这会为接下来的调试工作节省时间。同时也观察 `errno` 被设置时它如何打印错误信息。同样，这也可以节省你调试的时间。

C预处理器如何扩展宏

现在我会向你简单介绍一些预处理器的工作原理，让你知道这些宏是如何工作的。我会拆分 `dbg.h` 中阿最复杂的宏并且让你运行 `cpp` 来让你观察它实际上是如何工作的。

假设我有一个函数叫做 `dosomething()` ，执行成功是返回0，发生错误时返回-1。每次我调用 `dosomething` 的时候，我都要检查错误码，所以我将代码写成这样：

```c
int rc = dosomething();

if(rc != 0) {
    fprintf(stderr, "There was an error: %s\n", strerror());
    goto error;
}
```

我想使用预处理器做的是，将这个 `if` 语句封装为更可读并且便于记忆的一行代码。于是可以使用这个 `check` 来执行 `dbg.h` 中的宏所做的事情：

```c
int rc = dosomething();
check(rc == 0, "There was an error.");
```

这样更加简洁，并且恰好解释了所做的事情：检查函数是否正常工作，如果没有就报告错误。我们需要一些特别的预处理器“技巧”来完成它，这些技巧使预处理器作为代码生成工具更加易用。再次看看 `check` 和 `log_err` 宏：

```
#define log_err(M, ...) fprintf(stderr, "[ERROR] (%s:%d: errno: %s) " M "\n", __FILE__, __LINE__, clean_errno(), ##__VA_ARGS__)
#define check(A, M, ...) if(!(A)) { log_err(M, ##__VA_ARGS__); errno=0; goto error; }
```

第一个宏，`log_err` 更简单一些，只是将它自己替换为 `fprintf` 对 `stderr` 的调用。这个宏唯一的技巧性部分就是在 `log_err(M, ...)` 的定义中使用 `...` 。它所做的是让你向宏传入可变参数，从而传入 `fprintf` 需要接收的参数。它们是如何注入 `fprintf` 的呢？观察末尾的 `##__VA_ARGS__` ，它告诉预处理器将 `...` 所在位置的参数注入到 `fprintf` 调用的相应位置。于是你可以像这样调用了：

```
log_err("Age: %d, name: %s", age, name);
```

`age, name` 参数就是 `...` 所定义的部分，这些参数会被注入到 `fprintf` 中，输出会变成：

```
fprintf(stderr, "[ERROR] (%s:%d: errno: %s) Age %d: name %d\n",
    __FILE__, __LINE__, clean_errno(), age, name);
```

看到末尾的 `age, name` 了吗？这就是 `...` 和 `##__VA_ARGS__` 的工作机制，在调用其它变参宏（或者函数）的时候它会起作用。观察 `check` 宏调用 `log_err` 的方式，它也是用了 `...` 和 `##__VA_ARGS__` 。这就是传递整个 `printf` 风格的格式字符串给 `check` 的途径，它之后会传给 `log_err` ，二者的机制都像 `printf` 一样。

下一步是学习 `check` 如何为错误检查构造 `if` 语句，如果我们剖析 `log_err` 的用法，我们会得到：

```
if(!(A)) { errno=0; goto error; }
```

它的意思是，如果 `A` 为假，则重置 `errno` 并且调用 `error` 标签。`check` 宏会被上述 `if` 语句·替换，所以如果我们手动扩展 `check(rc == 0, "There was an error.")` ，我们会得到：

```
if(!(rc == 0)) {
    log_err("There was an error.");
    errno=0;
    goto error;
}
```

在这两个宏的展开过程中，你应该了解了预处理器会将宏替换为它的定义的扩展版本，并且递归地来执行这个步骤，扩展宏定义中的宏。预处理器是个递归的模板系统，就像我之前提到的那样。它的强大来源于使用参数化的代码来生成整个代码块，这使它成为便利的代码生成工具。

下面只剩一个问题了：为什么不像 `die` 一样使用函数呢？原因是需要在错误处理时使用 `file:line` 的数值和 `goto` 操作。如果你在函数在内部执行这些，你不会得到错误真正出现位置的行号，并且 `goto` 的实现也相当麻烦。

另一个原因是，如果你编写原始的 `if` 语句，它看起来就像是你代码中的其它的 `if` 语句，所以它看起来并不像一个错误检查。通过将 `if` 语句包装成 `check` 宏，就会使这一错误检查的逻辑更清晰，而不是主控制流的一部分。

最后，C预处理器提供了条件编译部分代码的功能，所以你可以编写只在构建程序的开发或调试版本时需要的代码。你可以看到这在 `dbg.h` 中已经用到了， `debug` 宏的主体部分只被编译器用到。如果没有这个功能，你需要多出一个 `if` 语句来检查是否为“调试模式”，也浪费了CPU资源来进行没有必要的检查。

附加题

- 将 `#define NDEBUG` 放在文件顶端来消除所有调试信息。
- 撤销上面添加的一行，并在 `MakeFile` 顶端将 `-D NDEBUG` 添加到 `CFLAGS` ，之后重新编译来达到同样效果。
- 修改日志宏，使之包含函数名称和 `file:line` 。

练习21：高级数据类型和控制结构

> 原文：Exercise 21: Advanced Data Types And Flow Control
>
> 译者：飞龙

这个练习是C语言中所有可用的数据类型和控制结构的摘要。它也可以作为一份参考在补完你的知识，并且不含有任何代码。我会通过创建教学卡片的方式，让你记住一些信息，所以你会在脑子里记住所有重要的概念。

这个练习非常有用，你应该花至少一周的时间来巩固内容并且补全这里所没有的元素。你应学出每个元素是什么意思，以及编写程序来验证你得出的结论。

可用的数据类型

`int`

储存普通的整数，默认为32位大小。

> 译者注：`int`在32或64位环境下为32位，但它不应该被看作平台无关的。如果需要用到平台无关的定长整数，请使用`int(n)_t`。

`double`

储存稍大的浮点数。

`float`

储存稍小的浮点数。

`char`

储存单字节字符。

`void`

表示“无类型”，用于声明不返回任何东西的函数，或者所指类型不明的指针，例如`void *thing`。

`enum`

枚举类型，类似于整数，也可转换为整数，但是通过符号化的名称访问或设置。当`switch`语句中没有覆盖到所有枚举的元素时，一些编译器会发出警告。

类型修饰符

`unsigned`

修改类型，使它不包含任何负数，同时上界变高。

`signed`

可以储存正数和负数，但是上界会变为（大约）一半，下界变为和上界（大约）等长。

> 译者注：符号修饰符只对 `char` 和 `*** int` 有效。 `*** int` 默认为 `signed` ，而 `char` 根据具体实现，可以默认为 `signed` ，也可以为 `unsigned` 。

`long`

对该类型使用较大的空间，使它能存下更大的数，通常使当前大小加倍。

`short`

对该类型使用较小的空间，使它储存能力变小，但是占据空间也变成一半。

类型限定符

`const`

表示变量在初始化后不能改变。

`volatile`

表示会做最坏的打算，编译器不会对它做任何优化。通常仅在对变量做一些奇怪的事情时，才会用到它。

`register`

强制让编译器将这个变量保存在寄存器中，并且也可以无视它。目前的编译器更善于处理在哪里存放变量，所以应该只在确定这样会提升性能时使用它。

类型转换

C使用了一种“阶梯形类型提升”的机制，它会观察运算符两边的变量，并且在运算之前将较小边的变量转换为较大边。这个过程按照如下顺序：

- long double
- double
- float
- long long
- long
- int (short, char)

> 译者注： `short` 和 `char` 会在运算之前转换成 `int` 。同种类型的 `unsigned` 和 `signed` 运算， `signed` 保持字节不变转换成 `unsigned` 。

类型大小

`stdint.h` 为定长的整数类型定义了一些 `typedef` ，同时也有一些用于这些类型的宏。这比老的 `limits.h` 更加易于使用，因为它是不变的。这些类型如下：

`int8_t`

8位符号整数。

`uint8_t`

8位无符号整数。

`int16_t`

16位符号整数。

`uint16_t`

16位无符号整数。

`int32_t`

32位符号整数。

`uint32_t`

32位无符号整数。

`int64_t`

64位符号整数。

`uint64_t`

64位无符号整数。

> 译者注：当用于对类型大小有要求的特定平台时，可以使用这些类型。如果你怕麻烦，不想处理平台相关类型的今后潜在的扩展的话，也可以使用这些类型。

下面的模式串为 `(u)int(BITS)_t` ，其中前面的 `u` 代表 `unsigned` ， `BITS` 是所占位数的大小。这些模式串返回了这些类型的最大（或最小）值。

`INT(N)_MAX`

`N` 位符号整数的最大正值，例如 `INT16_MAX` 。

`INT(N)_MIN`

`N` 位符号整数的最小负值。

`UINT(N)_MAX`

`N` 位无符号整数的最大正值。为什么不定义其最小值，是因为最小值是0，不可能出现负值。

> 警告
>
> 要注意，不要从字面上在任何头文件中去找`INT(N)_MAX`的定义。这里的`N`应该为特定整数，比如8、16、32、64，甚至可能是128。我在这个练习中使用了这个记法，就不需要显式写出每一个不同的组合了。

在`stdint.h`中，对于`size_t`类型和足够存放指针的整数也有一些宏定义，以及其它便捷类型的宏定义。编译器至少要保证它们为某一大小，并允许它们为更大的大小。

`int_least(N)_t`

至少`N`位的整数。

`uint_least(N)_t`

至少`N`位的无符号整数。

`INT_LEAST(N)_MAX`

`int_least(N)_t`类型的最大值。

`INT_LEAST(N)_MIN`

`int_least(N)_t`类型的最小值。

`UINT_LEAST(N)_MAX`

`uint_least(N)_t`的最大值。

`int_fast(N)_t`

与`int_least(N)_t`相似，但是是至少`N`位的“最快”整数。

`uint_fast(N)_t`

至少`N`位的“最快”无符号整数。

`INT_FAST(N)_MAX`

`int_fast(N)_t`的最大值。

`INT_FAST(N)_MIN`

`int_fast(N)_t`的最小值。

`UINT_FAST(N)_MAX`

`uint_fast(N)_t`的最大值。

`intptr_t`

足够存放指针的符号整数。

`uintptr_t`

足够存放指针的无符号整数。

`INTPTR_MAX`

`intptr_t` 的最大值。

`INTPTR_MIN`

`intptr_t` 的最小值。

`UINTPTR_MAX`

`uintptr_t` 的最大值。

`intmax_t`

系统中可能的最大尺寸的整数类型。

`uintmax_t`

系统中可能的最大尺寸的无符号整数类型。

`INTMAX_MAX`

`intmax_t` 的最大值。

`INTMAX_MIN`

`intmax_t` 的最小值。

`UINTMAX_MAX`

`uintmax_t` 的最大值。

`PTRDIFF_MIN`

`ptrdiff_t` 的最小值。

`PTRDIFF_MAX`

`ptrdiff_t` 的最大值。

`SIZE_MAX`

`size_t` 的最大值。

可用的运算符

这是一个全面的列表，关于你可以在C中使用的全部运算符。这个列表中我会标明一些东西：

二元

该运算符有左右两个操作数：`X + Y`。

一元

该运算符作用于操作数本身 `-X` 。

前缀

该运算符出现在操作数之前： `++X` 。

后缀

通常和前缀版本相似，但是出现在操作数之后，并且意义不同： `X++` 。

三元

只有一个三元运算符，意思是“三个操作数”： `X ? Y : Z` 。

算数运算符

下面是基本的算数运算符，我将函数调用 `()` 放入其中因为它更接近“算数”运算。

`()`

函数调用。

二元 `*`

乘法。

`/`

除法。

二元 `+`

加法。

一元 `+`

无变化。

后缀 `++`

读取变量然后自增。

前缀 `++`

自增变量然后读取。

后缀 `--`

读取变量然后自减。

前缀 `--`

自减变量然后读取。

二元 `-`

减法。

一元 `-`

取反，可用于表示负数。

数据运算

它们用于以不同方式和形式访问数据。

`->`

结构体指针的成员访问。一元 `*` 和 `.` 运算符的复合。

`.`

结构体值的成员访问。

`[]`

取数组下标。二元 `+` 和一元 `*` 运算符的复合。

`sizeof`

取类型或变量大小。

一元 `&`

取地址。

一元 `*`

取值（提领地址）。

逻辑运算符

它们用于测试变量的等性和不等性。

`!=`

不等于。

`<`

小于。

`<=`

小于等于。

`==`

等于（并不是赋值）。

`>`

大于。

`>=`

大于等于。

位运算符

它们更加高级，用于修改整数的原始位。

二元 `&`

位与。

`<<`

左移。

`>>`

右移。

`^`

位异或。

`|`

位或。

`~`

取补（翻转所有位）。

布尔运算符。

用于真值测试，仔细学习三元运算符，它非常有用。

`!`

取非。

`&&`

与。

`||`

或。

`?:`

三元真值测试，`X ? Y : Z` 读作“若X则Y否则Z”。

赋值运算符

复合赋值运算符在赋值同时执行运算。大多数上面的运算符都可以组成复合赋值运算符。

`=`

赋值。

`%=`

取余赋值。

`&=`

位与赋值。

`*=`

乘法赋值。

`+=`

加法赋值。

`-=`

减法赋值。

`/=`

除法赋值。

`<<=`

左移赋值。

`>>=`

右移赋值。

`^=`

位异或赋值。

`|=`

位或赋值。

可用的控制结构

下面是一些你没有接触过的控制结构：

`do-while`

`do { ... } while(X);` 首先执行花括号中的代码，之后再跳出前测试 `X` 表达式。

`break`

放在循环中用于跳出循环。

`continue`

跳到循环尾。

`goto`

跳到你已经放置 `label` 的位置，你已经在 `dbg.h` 中看到它了，用于跳到 `error` 标签。

附加题

- 阅读 `stdint.h` 或它的描述，写出所有可能出现的大小定义。
- 查询本练习的每一项，写出它在代码中的作用。上网浏览资料来研究它如何正确使用。
- 将这些信息做成教学卡片，每天看上15分钟来记住它们。
- 创建一个程序，打印出每个类型的示例，并验证你的研究结果是否正确。

练习22：栈、作用域和全局

> 原文：Exercise 22: The Stack, Scope, And Globals
>
> 译者：飞龙

许多人在开始编程时，对“作用域”这个概念都不是很清楚。起初它来源于系统栈的使用方式（在之前提到过一些），以及它用于临时变量储存的方式。这个练习中，我们会通过学习栈数据结构如何工作来了解作用域，然后再来看看现代C语言处理作用域的方式。

这个练习的眞正目的是了解一些比较麻烦的东西在C中如何存储。当一个人没有掌握作用域的概念时，它几乎也不能理解变量在哪里被创建，存在以及销毁。一旦你知道了这些，作用域的概念会变得易于理解。

这个练习需要如下三个文件：

`ex22.h`

用于创建一些外部变量和一些函数的头文件。

`ex22.c`

它并不像通常一样，是包含 `main` 的源文件，而是含有一些 `ex22.h` 中声明的函数和变量，并且会变成 `ex22.o` 。

`ex22_main.c`

`main` 函数实际所在的文件，它会包含另外两个文件，并演示了它们包含的东西以及其它作用域概念。

ex22.h 和 ex22.c

你的第一步是创建你自己的 `ex22.h` 头文件，其中定义了所需的函数和“导出”变量。

```c
#ifndef _ex22_h
#define _ex22_h

// makes THE_SIZE in ex22.c available to other .c files
extern int THE_SIZE;

// gets and sets an internal static variable in ex22.c
int get_age();
void set_age(int age);

// updates a static variable that's inside update_ratio
double update_ratio(double ratio);

void print_size();

#endif
```

最重要的事情是`extern int THE_SIZE`的用法，我将会在你创建完`ex22.c`之后解释它：

```c
#include <stdio.h>
#include "ex22.h"
#include "dbg.h"

int THE_SIZE = 1000;

static int THE_AGE = 37;

int get_age()
{
    return THE_AGE;
}

void set_age(int age)
{
    THE_AGE = age;
}


double update_ratio(double new_ratio)
{
    static double ratio = 1.0;

    double old_ratio = ratio;
    ratio = new_ratio;

    return old_ratio;
}

void print_size()
{
    log_info("I think size is: %d", THE_SIZE);
}
```

这两个文件引入了一些新的变量储存方式：

`extern`

这个关键词告诉编译器“这个变量已存在，但是他在别的‘外部区域’里”。通常它的意思是一个 `.c` 文件要用到另一个 `.c` 文件中定义的变量。这种情况下，我们可以说 `ex22.c` 中的 `THE_SIZE` 变量能被 `ex22_main.c` 访问到。

`static` （文件）

这个关键词某种意义上是 `extern` 的反义词，意思是这个变量只能在当前的 `.c` 文件中使用，程序的其它部分不可访问。要记住文件级别的 `static` （比如这里的 `THE_AGE` ）和其它位置不同。

`static` （函数）

如果你使用`static`在函数中声明变量，它和文件中的`static`定义类似，但是只能够在该函数中访问。它是一种创建某个函数的持续状态的方法，但事实上它很少用于现代的C语言，因为它们很难和线程一起使用。

在上面的两个文件中，你需要理解如下几个变量和函数：

`THE_SIZE`

这个你使用`extern`声明的变量将会在`ex22_main.c`中用到。

`get_age`和`set_age`

它们用于操作静态变量`THE_AGE`，并通过函数将其暴露给程序的其它部分。你不能够直接访问到`THE_AGE`，但是这些函数可以。

`update_ratio`

它生成新的`ratio`值并返回旧的值。它使用了函数级的静态变量`ratio`来跟踪`ratio`当前的值。

`print_size`

打印出`ex22.c`所认为的`THE_SIZE`的当前值。

ex22_main.c

一旦你写完了上面那些文件，你可以接着编程`main`函数，它会使用所有上面的文件并且演示了一些更多的作用域转换：

```
#include "ex22.h"
#include "dbg.h"

const char *MY_NAME = "Zed A. Shaw";

void scope_demo(int count)
{
    log_info("count is: %d", count);

    if(count > 10) {
        int count = 100;  // BAD! BUGS!

        log_info("count in this scope is %d", count);
    }

    log_info("count is at exit: %d", count);

    count = 3000;

    log_info("count after assign: %d", count);
}
```

```c
int main(int argc, char *argv[])
{
    // test out THE_AGE accessors
    log_info("My name: %s, age: %d", MY_NAME, get_age());

    set_age(100);

    log_info("My age is now: %d", get_age());

    // test out THE_SIZE extern
    log_info("THE_SIZE is: %d", THE_SIZE);
    print_size();

    THE_SIZE = 9;

    log_info("THE SIZE is now: %d", THE_SIZE);
    print_size();

    // test the ratio function static
    log_info("Ratio at first: %f", update_ratio(2.0));
    log_info("Ratio again: %f", update_ratio(10.0));
    log_info("Ratio once more: %f", update_ratio(300.0));

    // test the scope demo
    int count = 4;
    scope_demo(count);
    scope_demo(count * 20);

    log_info("count after calling scope_demo: %d", count);

    return 0;
}
```

我会把这个文件逐行拆分，你应该能够找到我提到的每个变量在哪里定义。

ex22_main.c:4

使用了 `const` 来创建常量，它可用于替代 `define` 来创建常量。

ex22_main.c:6

一个简单的函数，演示了函数中更多的作用域问题。

ex22_main.c:8

在函数顶端打印出 `count` 的值。

ex22_main.c:10

`if` 语句会开启一个新的作用域区块，并且在其中创建了另一个 `count` 变量。这个版本的 `count` 变量是一个全新的变量。 `if` 语句就好像开启了一个新的“迷你函数”。

ex22_main.c:11

`count` 对于当前区块是局部变量，实际上不同于函数参数列表中的参数。

ex22_main.c:13

将它打印出来，所以你可以在这里看到100，并不是传给 `scope_demo` 的参数。

ex22_main.c:16

这里是最难懂得部分。你在两部分都有 `count` 变量，一个数函数参数，另一个是 `if` 语句中。 `if` 语句创建了新的代码块，所以11行的 `count` 并不影响同名的参数。这一行将其打印出来，你会看到它打印了参数的值而不是100。

ex22_main.c:18-20

之后我将 `count` 参数设为3000并且打印出来，这里演示了你也可以修改函数参数的值，但并不会影响变量的调用者版本。

确保你浏览了整个函数，但是不要认为你已经十分了解作用娱乐。如果你在一个代码块中（比如 `if` 或 `while` 语句）创建了一些变量，这些变量是全新的变量，并且只在这个代码块中存在。这是至关重要的东西，也是许多bug的来源。我要强调你应该在这里花一些时间。

`ex22_main.c` 的剩余部分通过操作和打印变量演示了它们的全部。

ex22_main.c:26

打印出 `MY_NAME` 的当前值，并且使用 `get_age` 读写器从 `ex22.c` 获取 `THE_AGE` 。

ex22_main.c:27-30

使用了 `ex22.c` 中的 `set_age` 来修改并打印 `THE_AGE` 。

ex22_main.c:33-39

接下来我对 `ex22.c` 中的 `THE_SIZE` 做了相同的事情，但这一次我直接访问了它，并且同时演示了它实际上在那个文件中已经修改了，还使用 `print_size` 打印了它。

ex22_main.c:42-44

展示了 `update_ratio` 中的 `ratio` 在两次函数调用中如何保持了它的值。

ex22_main.c:46-51

最后运行 `scope_demo` ，你可以在实例中观察到作用域。要注意到的关键点是， `count` 局部变量在调用后保持不变。你将它像一个变量一样传入函数，它一定不会发生改变。要想达到目的你需要我们的老朋友指针。如果你将指向 `count` 的指针传入函数，那么函数就会持有它的地址并且能够改变它。

上面解释了这些文件中所发生的事情，但是你应该跟踪它们，并且确保在你学习的过程中明白了每个变量都在什么位置。

你会看到什么

这次我想让你手动构建这两个文件，而不是使用你的 `Makefile` 。于是你可以看到它们实际上如何被编译器放到一起。这是你应该做的事情，并且你应该看到如下输出：

```
$ cc -Wall -g -DNDEBUG   -c -o ex22.o ex22.c
$ cc -Wall -g -DNDEBUG    ex22_main.c ex22.o   -o ex22_main
$ ./ex22_main
[INFO] (ex22_main.c:26) My name: Zed A. Shaw, age: 37
[INFO] (ex22_main.c:30) My age is now: 100
[INFO] (ex22_main.c:33) THE_SIZE is: 1000
[INFO] (ex22.c:32) I think size is: 1000
[INFO] (ex22_main.c:38) THE SIZE is now: 9
[INFO] (ex22.c:32) I think size is: 9
[INFO] (ex22_main.c:42) Ratio at first: 1.000000
[INFO] (ex22_main.c:43) Ratio again: 2.000000
[INFO] (ex22_main.c:44) Ratio once more: 10.000000
[INFO] (ex22_main.c:8) count is: 4
[INFO] (ex22_main.c:16) count is at exit: 4
[INFO] (ex22_main.c:20) count after assign: 3000
[INFO] (ex22_main.c:8) count is: 80
[INFO] (ex22_main.c:13) count in this scope is 100
[INFO] (ex22_main.c:16) count is at exit: 80
[INFO] (ex22_main.c:20) count after assign: 3000
[INFO] (ex22_main.c:51) count after calling scope_demo: 4
```

确保你跟踪了每个变量是如何改变的，并且将其匹配到所输出的那一行。我使用了 `dbg.h` 的 `log_info` 来让你获得每个变量打印的具体行号，并且在文件中找到它用于跟踪。

作用域、栈和Bug

如果你正确完成了这个练习，你会看到有很多不同方式在C代码中放置变量。你可以使用 `extern` 或者访问类似 `get_age` 的函数来创建全局。你也可以在任何代码块中创建新的变量，它们在退出代码块之前会拥有自己的值，并且屏蔽掉外部的变量。你也可以响函数传递一个值并且修改它，但是调用者的变量版本不会发生改变。

需要理解的最重要的事情是，这些都可以造成bug。C中在你机器中许多位置放置和访问变量的能力会让你对它们所在的位置感到困扰。如果你不知道它们的位置，你就可能不能适当地管理它们。

下面是一些编程C代码时需要遵循的规则，可以让你避免与栈相关的bug：

- 不要隐藏某个变量，就像上面`scope_demo`中对`count`所做的一样。这可能会产生一些隐蔽的bug，你认为你改变了某个变量但实际上没有。
- 避免过多的全局变量，尤其是跨越多个文件。如果必须的话，要使用读写器函数，就像`get_age`。这并不适用于常量，因为它们是只读的。我是说对于`THE_SIZE`这种变量，如果你希望别人能够修改它，就应该使用读写器函数。
- 在你不清楚的情况下，应该把它放在堆上。不要依赖于栈的语义，或者指定区域，而是要直接使用`malloc`创建它。
- 不要使用函数级的静态变量，就像`update_ratio`。它们并不有用，而且当你想要使你的代码运行在多线程环境时，会有很大的隐患。对于良好的全局变量，它们也非常难于寻找。
- 避免复用函数参数，因为你搞不清楚仅仅想要复用它还是希望修改它的调用者版本。

如何使它崩溃

对于这个练习，崩溃这个程序涉及到尝试访问或修改你不能访问的东西。

- 试着从`ex22_main.c`直接访问`ex22.c`中的你不能访问变量。例如，你能不能获取`update_ratio`中的`ratio`？如果你用一个指针指向它会发生什么？
- 移除`ex22.h`的`extern`声明，来观察会得到什么错误或警告。
- 对不同变量添加`static`或者`const`限定符，之后尝试修改它们。

附加题

- 研究“值传递”和“引用传递”的差异，并且为二者编写示例。（译者注：C中没有引用传递，你可以搜索“指针传递”。）
- 使用指针来访问原本不能访问的变量。
- 使用`Valgrind`来观察错误的访问是什么样子。
- 编写一个递归调用并导致栈溢出的函数。如果不知道递归函数是什么的话，试着在`scope_demo`底部调用`scope_demo`本身，会形成一种循环。
- 重新编写`Makefile`使之能够构建这些文件。

练习23：认识达夫设备

> 原文：Exercise 23: Meet Duff's Device
>
> 译者：飞龙

这个练习是一个脑筋急转弯，我会向你介绍最著名的C语言黑魔法之一，叫做“达夫设备”，以“发明者”汤姆·达夫的名字命名。这一强大（或邪恶？）的代码中，几乎你学过的任何东西都被包装在一个小的结构中。弄清它的工作机制也是一个好玩的谜题。

> 注
>
> C的一部分乐趣来源于这种神奇的黑魔法，但这也是使C难以使用的地方。你最好能够了解这些技巧，因为他会带给你关于C语言和你计算机的深入理解。但是，你应该永远都不要使用它们，并总是追求简单易读的代码。

达夫设备由汤姆·达夫“发现”（或创造），它是一个C编译器的小技巧，本来不应该能够正常工作。我并不想告诉你做了什么，因为这是一个谜题，等着你来思考并尝试解决。你需要运行这段代码，之后尝试弄清它做了什么，以及为什么可以这样做。

```c
#include <stdio.h>
#include <string.h>
#include "dbg.h"


int normal_copy(char *from, char *to, int count)
{
    int i = 0;

    for(i = 0; i < count; i++) {
        to[i] = from[i];
    }

    return i;
}

int duffs_device(char *from, char *to, int count)
{
    {
        int n = (count + 7) / 8;

        switch(count % 8) {
            case 0: do { *to++ = *from++;
                        case 7: *to++ = *from++;
                        case 6: *to++ = *from++;
                        case 5: *to++ = *from++;
                        case 4: *to++ = *from++;
```

```
                        case 3: *to++ = *from++;
                        case 2: *to++ = *from++;
                        case 1: *to++ = *from++;
                    } while(--n > 0);
        }
    }

    return count;
}

int zeds_device(char *from, char *to, int count)
{
    {
        int n = (count + 7) / 8;

        switch(count % 8) {
            case 0:
            again: *to++ = *from++;

            case 7: *to++ = *from++;
            case 6: *to++ = *from++;
            case 5: *to++ = *from++;
            case 4: *to++ = *from++;
            case 3: *to++ = *from++;
            case 2: *to++ = *from++;
            case 1: *to++ = *from++;
                    if(--n > 0) goto again;
        }
    }

    return count;
}

int valid_copy(char *data, int count, char expects)
{
    int i = 0;
    for(i = 0; i < count; i++) {
        if(data[i] != expects) {
            log_err("[%d] %c != %c", i, data[i], expects);
            return 0;
        }
    }

    return 1;
}


int main(int argc, char *argv[])
{
    char from[1000] = {'a'};
    char to[1000] = {'c'};
    int rc = 0;
```

```
    // setup the from to have some stuff
    memset(from, 'x', 1000);
    // set it to a failure mode
    memset(to, 'y', 1000);
    check(valid_copy(to, 1000, 'y'), "Not initialized right.");

    // use normal copy to
    rc = normal_copy(from, to, 1000);
    check(rc == 1000, "Normal copy failed: %d", rc);
    check(valid_copy(to, 1000, 'x'), "Normal copy failed.");

    // reset
    memset(to, 'y', 1000);

    // duffs version
    rc = duffs_device(from, to, 1000);
    check(rc == 1000, "Duff's device failed: %d", rc);
    check(valid_copy(to, 1000, 'x'), "Duff's device failed copy."
);

    // reset
    memset(to, 'y', 1000);

    // my version
    rc = zeds_device(from, to, 1000);
    check(rc == 1000, "Zed's device failed: %d", rc);
    check(valid_copy(to, 1000, 'x'), "Zed's device failed copy."
);

    return 0;
error:
    return 1;
}
```

这段代码中我编写了三个版本的复制函数：

`normal_copy`

使用普通的 `for` 循环来将字符从一个数组复制到另一个。

`duffs_device`

这个就是称为“达夫设备”的脑筋急转弯，以汤姆·达夫的名字命名。这段有趣的邪恶代码应归咎于他。

`zeds_device`

“达夫设备”的另一个版本，其中使用了 `goto` 来让你发现一些线索，关于 `duffs_device` 中奇怪的 `do-while` 做了什么。

在往下学习之前仔细了解这三个函数，并试着自己解释代码都做了什么。

你会看到什么

这个程序没有任何输出，它只会执行并退出。你应当在`Valgrind`下运行它并确保没有任何错误。

解决谜题

首先需要了解的一件事，就是C对于它的一些语法是弱检查的。这就是你可以将`do-while`的一部分放入`switch`语句的一部分的原因，并且在其它地方的另一部分还可以正常工作。如果你观察带有`goto again`的我的版本，它实际上更清晰地解释了工作原理，但要确保你理解了这一部分是如何工作的。

第二件事是`switch`语句的默认贯穿机制可以让你跳到指定的`case`，并且继续运行直到`switch`结束。

最后的线索是`count % 8`以及顶端对`n`的计算。

现在，要理解这些函数的工作原理，需要完成下列事情：

- 将代码抄写在一张纸上。
- 当每个变量在`switch`之前初始化时，在纸的空白区域，把每个变量列在表中。
- 按照`switch`的逻辑模拟执行代码，之后再正确的`case`处跳出。
- 更新变量表，包括`to`、`from`和它们所指向的数组。
- 当你到达`while`或者我的`goto`时，检查你的变量，之后按照逻辑返回`do-while`顶端，或者`again`标签所在的地方。
- 继续这一手动的执行过程，更新变量，直到确定明白了代码如何运作。

为什么写成这样？

当你弄明白它的实际工作原理时，最终的问题是：为什么要把代码写成这样？这个小技巧的目的是手动编写“循环展开”。大而长的循环会非常慢，所以提升速度的一个方法就是找到循环中某个固定的部分，之后在循环中复制代码，序列化地展开。例如，如果你知道一个循环会执行至少20次，你就可以将这20次的内容直接写在源代码中。

达夫设备通过将循环展开为8个迭代块，来完成这件事情。这是个聪明的办法，并且可以正常工作。但是目前一个好的编译器也会为你完成这些。你不应该这样做，除非少数情况下你证明了它的确可以提升速度。

附加题

- 不要再这样写代码了。
- 查询维基百科的“达夫设备”词条，并且看看你能不能找到错误。将它与这里的版本对比，并且阅读文章来试着理解，为什么维基百科上的代码在你这里不能

正常工作，但是对于汤姆·达夫可以。

- 创建一些宏，来自动完成任意长度的这种设备。例如，你想创建32个 `case` 语句，并且不想手动把它们都写出来时，你会怎么办？你可以编写一次展开8个的宏吗？
- 修改 `main` 函数，执行一些速度检测，来看看哪个实际上更快。
- 查询 `memcpy` 、 `memmove` 和 `memset` ，并且也比较一下它们的速度。
- 不要再这样写代码了！

练习24：输入输出和文件

> 原文：Exercise 24: Input, Output, Files
>
> 译者：飞龙

你已经学会了使用`printf`来打印变量，这非常不错，但是还需要学习更多。这个练习中你会用到`fscanf`和`fgets`在结构体中构建关于一个人的信息。在这个关于读取输入的简介之后，你会得到C语言IO函数的完整列表。其中一些你已经见过并且使用过了，所以这个练习也是一个记忆练习。

```c
#include <stdio.h>
#include "dbg.h"

#define MAX_DATA 100

typedef enum EyeColor {
    BLUE_EYES, GREEN_EYES, BROWN_EYES,
    BLACK_EYES, OTHER_EYES
} EyeColor;

const char *EYE_COLOR_NAMES[] = {
    "Blue", "Green", "Brown", "Black", "Other"
};

typedef struct Person {
    int age;
    char first_name[MAX_DATA];
    char last_name[MAX_DATA];
    EyeColor eyes;
    float income;
} Person;


int main(int argc, char *argv[])
{
    Person you = {.age = 0};
    int i = 0;
    char *in = NULL;

    printf("What's your First Name? ");
    in = fgets(you.first_name, MAX_DATA-1, stdin);
    check(in != NULL, "Failed to read first name.");

    printf("What's your Last Name? ");
    in = fgets(you.last_name, MAX_DATA-1, stdin);
    check(in != NULL, "Failed to read last name.");

    printf("How old are you? ");
```

```c
    int rc = fscanf(stdin, "%d", &you.age);
    check(rc > 0, "You have to enter a number.");

    printf("What color are your eyes:\n");
    for(i = 0; i <= OTHER_EYES; i++) {
        printf("%d) %s\n", i+1, EYE_COLOR_NAMES[i]);
    }
    printf("> ");

    int eyes = -1;
    rc = fscanf(stdin, "%d", &eyes);
    check(rc > 0, "You have to enter a number.");

    you.eyes = eyes - 1;
    check(you.eyes <= OTHER_EYES && you.eyes >= 0, "Do it right,
 that's not an option.");

    printf("How much do you make an hour? ");
    rc = fscanf(stdin, "%f", &you.income);
    check(rc > 0, "Enter a floating point number.");

    printf("----- RESULTS -----\n");

    printf("First Name: %s", you.first_name);
    printf("Last Name: %s", you.last_name);
    printf("Age: %d\n", you.age);
    printf("Eyes: %s\n", EYE_COLOR_NAMES[you.eyes]);
    printf("Income: %f\n", you.income);

    return 0;
error:

    return -1;
}
```

这个程序非常简单，并且引入了叫做 `fscanf` 的函数，意思是“文件的格式化输入”。 `scanf` 家族的函数是 `printf` 的反转版本。 `printf` 用于以某种格式打印数据，然而 `scanf` 以某种格式读取（或者扫描）输入。

文件开头没有什么新的东西，所以下面只列出 `main` 所做的事情：

ex24.c:24-28

创建所需的变量。

ex24.c:30-32

使用 `fgets` 函数获取名字，它从输入读取字符串（这个例子中是 `stdin` ），但是确保它不会造成缓冲区溢出。

ex24.c:34-36

对 `you.last_name` 执行相同操作，同样使用了 `fgets` 。

ex24.c:38-39

使用 `fscanf` 来从 `stdin` 读取整数，并且将其放到 `you.age` 中。你可以看到，其中使用了和 `printf` 相同格式的格式化字符串。你也应该看到传入了 `you.age` 的地址，便于 `fscnaf` 获得它的指针来修改它。这是一个很好的例子，解释了使用指向数据的指针作为“输出参数”。

ex24.c:41-45

打印出用于眼睛颜色的所有可选项，并且带有 `EyeColor` 枚举所匹配的数值。

ex24.c:47-50

再次使用了 `fscanf` ，从 `you.eyes` 中获取数值，但是保证了输入是有效的。这非常重要，因为用户可以输入一个超出 `EYE_COLOR_NAMES` 数组范围的值，并且会导致段错误。

ex24.c:52-53

获取 `you.income` 的值。

ex24.c:55-61

将所有数据打印出来，便于你看到它们是否正确。要注意 `EYE_COLOR_NAMES` 用于打印 `EyeColor` 枚举值实际上的名字。

你会看到什么

当你运行这个程序时，你应该看到你的输入被适当地转换。你应该尝试给它非预期的输入，看看程序是怎么预防它的。

```
$ make ex24
cc -Wall -g -DNDEBUG    ex24.c   -o ex24
$ ./ex24
What's your First Name? Zed
What's your Last Name? Shaw
How old are you? 37
What color are your eyes:
1) Blue
2) Green
3) Brown
4) Black
5) Other
> 1
How much do you make an hour? 1.2345
----- RESULTS -----
First Name: Zed
Last Name: Shaw
Age: 37
Eyes: Blue
Income: 1.234500
```

如何使它崩溃

这个程序非常不错，但是这个练习中真正重要的部分是， `scanf` 如何发生错误。对于简单的数值转换没有问题，但是对于字符串会出现问题，因为 `scanf` 在你读取之前并不知道缓冲区有多大。类似于 `gets` 的函数（并不是 `fgets` ，不带 `f` 的版本）也有一个我们已经避免的问题。它并不是道输入缓冲区有多大，并且可能会使你的程序崩溃。

要演示 `fscanf` 和字符串的这一问题，需要修改使用 `fgets` 的那一行，使它变成 `fscanf(stdin, "%50s", you.first_name)` ，并且尝试再次运行。你会注意到，它读取了过多的内容，并且吃掉了你的回车键。这并不是你期望它所做的，你应该使用 `fgets` 而不是去解决古怪的 `scanf` 问题。

接下来，将 `fgets` 改为 `gets` ，接着使用 `valgrind` 来执行 `valgrind ./ex24 < /dev/urandom` ，往你的程序中输入一些垃圾字符串。这叫做对你的程序进行“模糊测试”，它是一种不错的方法来发现输入错误。这个例子中，你需要从 `/dev/urandom` 文件来输入一些垃圾，并且观察它如何崩溃。在一些平台上你需要执行数次，或者修改 `MAX_DATA` 来使其变小。

`gets` 函数非常糟糕，以至于一些平台在程序运行时会警告你使用了 `gets` 。你应该永远避免使用这个函数。

最后，找到 `you.eyes` 输入的地方，并移除对其是否在正确范围内的检查。然后，为它输入一个错误的数值，比如-1或者1000。在 `Valgrind` 执行这些操作，来观察会发生什么。

> 译者注：根据最新的C11标准，对于输入函数，你应该总是使用`_s`后缀的安全版本。对于向字符串的输出函数，应该总是使用C99中新增的带`n`的版本，例如`snprintf`。如果你的编译器支持新版本，就不应该使用旧版本的不安全函数。

IO函数

这是一个各种IO函数的简单列表。你应该查询每个函数并为其创建速记卡，包含函数名称，功能和它的任何变体。

- `fscanf`
- `fgets`
- `fopen`
- `freopen`
- `fdopen`
- `fclose`
- `fcloseall`
- `fgetpos`
- `fseek`
- `ftell`
- `rewind`
- `fprintf`
- `fwrite`
- `fread`

过一遍这些函数，并且记住它们的不同变体和它们的功能。例如，对于`fscanf`的卡片，上面应该有`scanf`、`sscanf`、`vscanf`，以及其它。并且在背面写下每个函数所做的事情。

最后，为了获得这些卡片所需的信息，使用`man`来阅读它的帮助。例如，`fscanf`帮助页由`man fscanf`得到。

附加题

- 将这个程序重写为不需要`fscanf`的版本。你需要使用类似于`atoi`的函数来将输入的字符串转换为数值。
- 修改这个程序，使用`scanf`来代替`fscanf`，并观察有什么不同。
- 修改程序，是输入的名字不包含任何换行符和空白字符。
- 使用`scanf`编写函数，按照文件名读取文件内容，每次读取单个字符，但是不要越过（文件和缓冲区的）末尾。使这个函数接受字符串大小来更加通用，并且确保无论什么情况下字符串都以`'\0'`结尾。

练习25：变参函数

> 原文：Exercise 25: Variable Argument Functions
>
> 译者：飞龙

在C语言中，你可以通过创建“变参函数”来创建你自己的 `printf` 或者 `scanf` 版本。这些函数使用 `stdarg.h` 头，它们可以让你为你的库创建更加便利的接口。它们对于创建特定类型的“构建”函数、格式化函数和任何用到可变参数的函数都非常实用。

理解“变参函数”对于C语言编程并不必要，我在编程生涯中也只有大约20次用到它。但是，理解变参函数如何工作有助于你对它的调试，并且让你更加了解计算机。

```
/** WARNING: This code is fresh and potentially isn't correct ye
t. */

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include "dbg.h"

#define MAX_DATA 100

int read_string(char **out_string, int max_buffer)
{
    *out_string = calloc(1, max_buffer + 1);
    check_mem(*out_string);

    char *result = fgets(*out_string, max_buffer, stdin);
    check(result != NULL, "Input error.");

    return 0;

error:
    if(*out_string) free(*out_string);
    *out_string = NULL;
    return -1;
}

int read_int(int *out_int)
{
    char *input = NULL;
    int rc = read_string(&input, MAX_DATA);
    check(rc == 0, "Failed to read number.");

    *out_int = atoi(input);
```

```
    free(input);
    return 0;

error:
    if(input) free(input);
    return -1;
}

int read_scan(const char *fmt, ...)
{
    int i = 0;
    int rc = 0;
    int *out_int = NULL;
    char *out_char = NULL;
    char **out_string = NULL;
    int max_buffer = 0;

    va_list argp;
    va_start(argp, fmt);

    for(i = 0; fmt[i] != '\0'; i++) {
        if(fmt[i] == '%') {
            i++;
            switch(fmt[i]) {
                case '\0':
                    sentinel("Invalid format, you ended with %%."
);
                    break;

                case 'd':
                    out_int = va_arg(argp, int *);
                    rc = read_int(out_int);
                    check(rc == 0, "Failed to read int.");
                    break;

                case 'c':
                    out_char = va_arg(argp, char *);
                    *out_char = fgetc(stdin);
                    break;

                case 's':
                    max_buffer = va_arg(argp, int);
                    out_string = va_arg(argp, char **);
                    rc = read_string(out_string, max_buffer);
                    check(rc == 0, "Failed to read string.");
                    break;

                default:
                    sentinel("Invalid format.");
            }
        } else {
            fgetc(stdin);
        }
```

```
        check(!feof(stdin) && !ferror(stdin), "Input error.");
    }

    va_end(argp);
    return 0;

error:
    va_end(argp);
    return -1;
}



int main(int argc, char *argv[])
{
    char *first_name = NULL;
    char initial = ' ';
    char *last_name = NULL;
    int age = 0;

    printf("What's your first name? ");
    int rc = read_scan("%s", MAX_DATA, &first_name);
    check(rc == 0, "Failed first name.");

    printf("What's your initial? ");
    rc = read_scan("%c\n", &initial);
    check(rc == 0, "Failed initial.");

    printf("What's your last name? ");
    rc = read_scan("%s", MAX_DATA, &last_name);
    check(rc == 0, "Failed last name.");

    printf("How old are you? ");
    rc = read_scan("%d", &age);

    printf("---- RESULTS ----\n");
    printf("First Name: %s", first_name);
    printf("Initial: '%c'\n", initial);
    printf("Last Name: %s", last_name);
    printf("Age: %d\n", age);

    free(first_name);
    free(last_name);
    return 0;
error:
    return -1;
}
```

这个程序和上一个练习很像，除了我编写了自己的 `scanf` 风格函数，它以我自己的方式处理字符串。你应该对 `main` 函数很清楚了，以及 `read_string` 和 `read_int` 两个函数，因为它们并没有做什么新的东西。

这里的变参函数叫做 `read_scan` ，它使用了 `va_list` 数据结构执行和 `scanf` 相同的工作，并支持宏和函数。下面是它的工作原理：

- 我将函数的最后一个参数设置为 `...` ，它向C表示这个函数在 `fmt` 参数之后接受任何数量的参数。我可以在它前面设置许多其它的参数，但是在它后面不能放置任何参数。
- 在设置完一些参数时，我创建了 `va_list` 类型的变量，并且使用 `va_list` 来为其初始化。这配置了 `stdarg.h` 中的这一可以处理可变参数的组件。
- 接着我使用了 `for` 循环，遍历 `fmt` 格式化字符串，并且处理了类似 `scanf` 的格式，但比它略简单。它里面只带有整数、字符和字符串。
- 当我碰到占位符时，我使用了 `switch` 语句来确定需要做什么。
- 现在，为了从 `va_list argp` 中获得遍历，我需要使用 `va_arg(argp, TYPE)` 宏，其中 `TYPE` 是我将要向参数传递的准确类型。这一设计的后果是你会非常盲目，所以如果你没有足够的变量传入，程序就会崩溃。
- 和 `scanf` 的有趣的不同点是，当它碰到 `'s'` 占位符时，我使用 `read_string` 来创建字符串。 `va_list argp` 栈需要接受两个函数：需要读取的最大尺寸，以及用于输出的字符串指针。 `read_string` 使用这些信息来执行实际工作。
- 这使 `read_scan` 比 `scan` 更加一致，因为你总是使用 `&` 提供变量的地址，并且合理地设置它们。
- 最后，如果它碰到了不在格式中的字符，它仅仅会读取并跳过，而并不关心字符是什么，因为它只需要跳过。

你会看到什么

当你运行程序时，会得到与下面详细的结果：

```
$ make ex25
cc -Wall -g -DNDEBUG    ex25.c   -o ex25
$ ./ex25
What's your first name? Zed
What's your initial? A
What's your last name? Shaw
How old are you? 37
---- RESULTS ----
First Name: Zed
Initial: 'A'
Last Name: Shaw
Age: 37
```

如何使它崩溃

这个程序对缓冲区溢出更加健壮，但是和 `scanf` 一样，它不能够处理输入的格式错误。为了使它崩溃，试着修改代码，把首先传入用于 `'%s'` 格式的尺寸去掉。同时试着传入多于 `MAX_DATA` 的数据，之后找到在 `read_string` 中不使用 `calloc` 的方法，并且修改它的工作方式。最后还有个问题是 `fgets` 会吃掉换行符，所以试着使用 `fgetc` 修复它，要注意字符串结尾应为 `'\0'` 。

附加题

- 再三检查确保你明白了每个 `out_` 变量的作用。最重要的是 `out_string` ，并且它是指针的指针。所以，理清当你设置时获取到的是指针还是内容尤为重要。
- 使用变参系统编写一个和 `printf` 相似的函数，重新编写 `main` 来使用它。
- 像往常一样，阅读这些函数/宏的手册页，确保知道了它在你的平台做了什么，一些平台会使用宏而其它平台会使用函数，还有一些平台会让它们不起作用。这完全取决于你所用的编译器和平台。

练习26：编写第一个真正的程序

> 原文：Exercise 26: Write A First Real Program
>
> 译者：飞龙

这本书你已经完成一半了，所以你需要做一个期中检测。期中检测中你需要重新构建一个我特地为本书编写的软件，叫做 `devpkg` 。随后你需要以一些方式扩展它，并且通过编写一些单元测试来改进代码。

> 注
>
> 我在一些你需要完成的练习之前编写了这个练习。如果你现在尝试这个练习，记住软件可能会含有一些bug，你可能由于我的错误会产生一些问题，也可能不知道需要什么来完成它。如果这样的话，通过help@learncodethehardway.org来告诉我，之后等待我写完其它练习。

什么是 `devpkg` ？

`devpkg` 是一个简单的C程序，可以用于安装其它软件。我特地为本书编写了它，作为一种方式来教你真正的软件是如何构建的，以及如何复用他人的库。它使用了一个叫做Apache可移植运行时（APR）的库，其中含有许多工作跨平台的便利的C函数，包括Windows。此外，它只是从互联网（或本地文件）抓取代码，并且执行通常的 `./configure ; make ; make install` 命令，每个程序员都用到过。

这个练习中，你的目标是从源码构建 `devpkg` ，完成我提供的每个挑战，并且使用源码来理解 `devpkg` 做了什么和为什么这样做。

我们打算创建什么

我们打算创建一个具有三个命令的工具：

devpkg -S

在电脑上安装新的软件。

devpkg -I

从URL安装软件。

devpkg -L

列出安装的所有软件。

devpkg -F

为手动构建抓取源代码。

devpkg -B

构建所抓取的源码代码并且安装它，即使它已经安装了。

我们想让`devpkg`能够接受几乎任何URL，判断项目的类型，下载，安装，以及注册已经安装的软件。我们也希望它能够处理一个简单的依赖列表，以便它能够安装项目所需的所有软件。

设计

为了完成这一目标，`devpkg`具有非常简单的设计：

使用外部命令

大多数工作都是通过类似于`curl`、`git`和`tar`的外部命令完成的。这样减少了`devpkg`所需的代码量。

简单的文件数据库

你可以轻易使它变得很复杂，但是一开始你需要完成一个简单的文件数据库，位于`/usr/local/.devpkg/db`，来跟踪已安装的软件。

`/usr/local`

同样你可以使它更高级，但是对于初学者来说，假设项目始终位于`/usr/local`中，它是大多数Unix软件的标准安装目录。

`configure; make; make install`

假设大多数软件可以通过`configure; make; make install`来安装，也许`configure`是可选的。如果软件不能通过这种方式安装，要么提供某种方式来修改命令，要么`devpkg`就可以无视它。

用户可以root

我们假设用于可以使用`sudo`来提升至root权限，除非他们直到最后才想root。

这会使我们的程序像当初设想的一样简单，并且对于它的功能来说已经足够了。之后你可以进一步修改它。

Apache 可移植运行时

你需要做的另外一件事情就是使用Apache可移植运行时（APR）来未完成这个练习获得一个可移植的工具集。APR并不是必要的，你也可以不用它，但是你需要写的代码就会非常多。我现在强制你使用APR，使你能够熟悉链接和使用其他的库。最后，APR也能在Windows上工作，所以你可以把它迁移到许多其它平台上。

你应该获取`apr-1.4.5`和`apr-util-1.3`的库，以及浏览在apr.apache.org主站上的文档。

下面是一个ShellScript，用于安装所需的所有库。你应该手动将它写到一个文件中，之后运行它直到APR安装好并且没有任何错误。

```
set -e

# go somewhere safe
cd /tmp

# get the source to base APR 1.4.6
curl -L -O http://archive.apache.org/dist/apr/apr-1.4.6.tar.gz

# extract it and go into the source
tar -xzvf apr-1.4.6.tar.gz
cd apr-1.4.6

# configure, make, make install
./configure
make
sudo make install

# reset and cleanup
cd /tmp
rm -rf apr-1.4.6 apr-1.4.6.tar.gz

# do the same with apr-util
curl -L -O http://archive.apache.org/dist/apr/apr-util-1.4.1.tar
.gz

# extract
tar -xzvf apr-util-1.4.1.tar.gz
cd apr-util-1.4.1

# configure, make, make install
./configure --with-apr=/usr/local/apr
# you need that extra parameter to configure because
# apr-util can't really find it because...who knows.

make
sudo make install

#cleanup
cd /tmp
rm -rf apr-util-1.4.1* apr-1.4.6*
```

我希望你输入这个脚本，因为这就是 `devpkg` 基本上所做的事情，只是带有了一些选项和检查项。实际上，你可以使用Shell以更少的代码来完成它，但是这对于一本C语言的书不是一个很好的程序。

简单运行这个脚本，修复它直到正常工作，就完成的所有库的安装，之后你需要完成项目的剩下部分。

项目布局

你需要创建一些简单的项目文件来起步。下面是我通常创建一个新项目的方法：

```
mkdir devpkg
cd devpkg
touch README Makefile
```

其它依赖

你应该已经安装了APR和APR-util，所以你需要一些更多的文件作为基本的依赖：

- 练习20中的 `dbg.h` 。
- 从http://bstring.sourceforge.net/下载的 `bstrlib.h` 和 `bstrlib.c` 。下载 `.zip` 文件，解压并且将这个两个文件拷贝到项目中。
- 运行 `make bstrlib.o` ，如果这不能正常工作，阅读下面的“修复 `bstring` ”指南。

> 注
>
> 在一些平台上 `bstring.c` 文件会出现下列错误：
>
> ```
> bstrlib.c:2762: error: expected declaration specifiers or '...' before numeric constant
> ```
>
> 这是由于作者使用了一个不好的定义，它在一些平台上不能工作。你需要修改第2759行的 `#ifdef __GNUC__` ，并把它改成：
>
> ```
> #if defined(__GNUC__) && !defined(__APPLE__)
> ```

之后在Mac OSX平台上就应该能够正常工作了。

做完上面这些后，你应该有了 `Makefile` ， `README` ， `dbg.h` ， `bstrlib.h` 和 `bstrlib.c` ，并做好了准备。

Makefile

我们最好从 `Makefile` 开始，因为它列出了项目如何构建，以及你会创建哪些源文件。

```
PREFIX?=/usr/local
CFLAGS=-g -Wall -I${PREFIX}/apr/include/apr-1  -I${PREFIX}/apr/i
nclude/apr-util-1
LDFLAGS=-L${PREFIX}/apr/lib -lapr-1 -pthread -laprutil-1

all: devpkg

devpkg: bstrlib.o db.o shell.o commands.o

install: all
	install -d $(DESTDIR)/$(PREFIX)/bin/
	install devpkg $(DESTDIR)/$(PREFIX)/bin/

clean:
	rm -f *.o
	rm -f devpkg
	rm -rf *.dSYM
```

比起之前看到过的，这并没有什么新东西，除了可能有些奇怪的 `?=` 语法，它表示“如果之前没有定义，就将 `PREFIX` 设置为该值”。

> 注
>
> 如果你使用了最近版本的Ubuntu，你会得到 `apr_off_t` 或 `off64_t` 的错误，之后需要向 `CFLAGS` 添加 `-D_LARGEFILE64_SOURCE=1` 。
>
> 所需的另一件事是，你需要向 `/etc/ld.conf.so.d/` 添加 `/usr/local/apr/lib` ，之后运行 `ldconfig` 使它能够选择正常的库。

源文件

我们可以从 `makefile` 中看到， `devpkg` 有四个依赖项，它们是：

`bstrlib.o`

由 `bstrlib.c` 和 `bstrlib.o` 产生，你已经将它们引入了。

`db.o`

由 `db.c` 和 `db.h` 产生，它包含了一个小型“数据库”程序集的代码。

`shell.o`

由 `shell.c` 和 `shell.h` 产生，包含一些函数，是类似 `curl` 的一些命令运行起来更容易。

`commands.o`

由 `commands.c` 和 `commands.h` 产生，包含了 `devpkg` 所需的所有命令并使它更易用。

`devpkg`

它不会显式提到，但是它是 `Makefile` 在这一部分的目标。它由 `devpkg.c` 产生，包含用于整个程序的 `main` 函数。

你的任务就是创建这些文件，并且输入代码并保证正确。

> 注
>
> 你读完这个描述可能会想，“Zed为什么那么聪明，坐着就能设计出来这些文件？！”我并不是用我强大的代码功力魔术般地把 `devpkg` 设计成这样。而是我做了这些：
>
> - 我编写了简单的 `README` 来获得如何构建项目的灵感。
> - 我创建了一个简单的bash脚本（就像你编写的那样）来理清所需的所有组件。
> - 我创建了一个 `.c` 文件，并且在它上面花了几天，酝酿并想出点子。
> - 接着我编写并调试程序，之后我将这一个大文件分成四个文件。
> - 做完这些之后，我重命名和优化了函数和数据结构，使它们在逻辑上更“美观”。
> - 最后，使新程序成功并以相同方式工作之后，我添加了一些新的特性，比如 `-F` 和 `-B` 选项。
>
> 你读到的这份列表是我打算教给你的，但不要认为这是我构建软件的通用方法。有时候我会事先知道主题，并且会做更多的规划。也有时我会编写一份规划并将它扔掉，之后再规划更好的版本。它完全取决于我的经验告诉我哪个比较好，或者我的灵感将我带到何处。
>
> 如果你碰到一个“专家”，它告诉你只有一个方法可以解决编程问题，那么它在骗你。要么它们实际使用了很多策略，要么他们并不足够好。

DB函数

程序中必须有个方法来记录已经安装的URL，列出这些URL，并且检查一些程序是否已安装以便跳过。我会使用一个简单、扁平化的文件数据库，以及 `bstrlib.h` 。

首先，创建 `db.h` 头文件，以便让你知道需要实现什么。

```
#ifndef _db_h
#define _db_h

#define DB_FILE "/usr/local/.devpkg/db"
#define DB_DIR "/usr/local/.devpkg"


int DB_init();
int DB_list();
int DB_update(const char *url);
int DB_find(const char *url);

#endif
```

之后实现 `db.c` 中的这些函数，在你编写它的时候，像之前一样使用 `make` 。

```
#include <unistd.h>
#include <apr_errno.h>
#include <apr_file_io.h>

#include "db.h"
#include "bstrlib.h"
#include "dbg.h"

static FILE *DB_open(const char *path, const char *mode)
{
    return fopen(path, mode);
}


static void DB_close(FILE *db)
{
    fclose(db);
}


static bstring DB_load()
{
    FILE *db = NULL;
    bstring data = NULL;

    db = DB_open(DB_FILE, "r");
    check(db, "Failed to open database: %s", DB_FILE);

    data = bread((bNread)fread, db);
    check(data, "Failed to read from db file: %s", DB_FILE);

    DB_close(db);
    return data;
```

```c
error:
    if(db) DB_close(db);
    if(data) bdestroy(data);
    return NULL;
}


int DB_update(const char *url)
{
    if(DB_find(url)) {
        log_info("Already recorded as installed: %s", url);
    }

    FILE *db = DB_open(DB_FILE, "a+");
    check(db, "Failed to open DB file: %s", DB_FILE);

    bstring line = bfromcstr(url);
    bconchar(line, '\n');
    int rc = fwrite(line->data, blength(line), 1, db);
    check(rc == 1, "Failed to append to the db.");

    return 0;
error:
    if(db) DB_close(db);
    return -1;
}


int DB_find(const char *url)
{
    bstring data = NULL;
    bstring line = bfromcstr(url);
    int res = -1;

    data = DB_load();
    check(data, "Failed to load: %s", DB_FILE);

    if(binstr(data, 0, line) == BSTR_ERR) {
        res = 0;
    } else {
        res = 1;
    }

error: // fallthrough
    if(data) bdestroy(data);
    if(line) bdestroy(line);

    return res;
}


int DB_init()
{
```

```
    apr_pool_t *p = NULL;
    apr_pool_initialize();
    apr_pool_create(&p, NULL);

    if(access(DB_DIR, W_OK | X_OK) == -1) {
        apr_status_t rc = apr_dir_make_recursive(DB_DIR,
                APR_UREAD | APR_UWRITE | APR_UEXECUTE |
                APR_GREAD | APR_GWRITE | APR_GEXECUTE, p);
        check(rc == APR_SUCCESS, "Failed to make database dir: %
s", DB_DIR);
    }

    if(access(DB_FILE, W_OK) == -1) {
        FILE *db = DB_open(DB_FILE, "w");
        check(db, "Cannot open database: %s", DB_FILE);
        DB_close(db);
    }

    apr_pool_destroy(p);
    return 0;

error:
    apr_pool_destroy(p);
    return -1;
}


int DB_list()
{
    bstring data = DB_load();
    check(data, "Failed to read load: %s", DB_FILE);

    printf("%s", bdata(data));
    bdestroy(data);
    return 0;

error:
    return -1;
}
```

挑战1：代码复查

在继续之前，仔细阅读这些文件的每一行，并且确保你以准确地输入了它们。通过逐行阅读代码来实践它。同时，跟踪每个函数调用，并且确保你使用了 `check` 来校验返回值。最后，在APR网站上的文档，或者bstrlib.h 或 bstrlib.c的源码中，查阅每个你不认识的函数。

Shell 函数

`devkpg` 的一个关键设计是，使用类似于 `curl` 、 `tar` 和 `git` 的外部工具来完成大部分的工作。我们可以找到在程序内部完成这些工作的库，但是如果我们只是需要这些程序的基本功能，这样就毫无意义。在Unix运行其它命令并不丢人。

为了完成这些，我打算使用 `apr_thread_proc.h` 函数来运行程序，但是我也希望创建一个简单的类“模板”系统。我会使用 `struct Shell` ，它持有所有运行程序所需的信息，但是在参数中有一些“空位”，我可以将它们替换成实际值。

观察 `shell.h` 文件来了解我会用到的结构和命令。你可以看到我使用 `extern` 来表明其他的 `.c` 文件也能访问到 `shell.c` 中定义的变量。

```
#ifndef _shell_h
#define _shell_h

#define MAX_COMMAND_ARGS 100

#include <apr_thread_proc.h>

typedef struct Shell {
    const char *dir;
    const char *exe;

    apr_procattr_t *attr;
    apr_proc_t proc;
    apr_exit_why_e exit_why;
    int exit_code;

    const char *args[MAX_COMMAND_ARGS];
} Shell;

int Shell_run(apr_pool_t *p, Shell *cmd);
int Shell_exec(Shell cmd, ...);

extern Shell CLEANUP_SH;
extern Shell GIT_SH;
extern Shell TAR_SH;
extern Shell CURL_SH;
extern Shell CONFIGURE_SH;
extern Shell MAKE_SH;
extern Shell INSTALL_SH;

#endif
```

确保你已经创建了 `shell.h` ，并且 `extern Shell` 变量的名字和数量相同。它们被 `Shell_run` 和 `Shell_exec` 函数用于运行命令。我定义了这两个函数，并且在 `shell.c` 中创建实际变量。

```
#include "shell.h"
#include "dbg.h"
#include <stdarg.h>
```

```
int Shell_exec(Shell template, ...)
{
    apr_pool_t *p = NULL;
    int rc = -1;
    apr_status_t rv = APR_SUCCESS;
    va_list argp;
    const char *key = NULL;
    const char *arg = NULL;
    int i = 0;

    rv = apr_pool_create(&p, NULL);
    check(rv == APR_SUCCESS, "Failed to create pool.");

    va_start(argp, template);

    for(key = va_arg(argp, const char *);
        key != NULL;
        key = va_arg(argp, const char *))
    {
        arg = va_arg(argp, const char *);

        for(i = 0; template.args[i] != NULL; i++) {
            if(strcmp(template.args[i], key) == 0) {
                template.args[i] = arg;
                break; // found it
            }
        }
    }

    rc = Shell_run(p, &template);
    apr_pool_destroy(p);
    va_end(argp);
    return rc;

error:
    if(p) {
        apr_pool_destroy(p);
    }
    return rc;
}

int Shell_run(apr_pool_t *p, Shell *cmd)
{
    apr_procattr_t *attr;
    apr_status_t rv;
    apr_proc_t newproc;

    rv = apr_procattr_create(&attr, p);
    check(rv == APR_SUCCESS, "Failed to create proc attr.");

    rv = apr_procattr_io_set(attr, APR_NO_PIPE, APR_NO_PIPE,
            APR_NO_PIPE);
```

```
    check(rv == APR_SUCCESS, "Failed to set IO of command.");

    rv = apr_procattr_dir_set(attr, cmd->dir);
    check(rv == APR_SUCCESS, "Failed to set root to %s", cmd->di
r);

    rv = apr_procattr_cmdtype_set(attr, APR_PROGRAM_PATH);
    check(rv == APR_SUCCESS, "Failed to set cmd type.");

    rv = apr_proc_create(&newproc, cmd->exe, cmd->args, NULL, at
tr, p);
    check(rv == APR_SUCCESS, "Failed to run command.");

    rv = apr_proc_wait(&newproc, &cmd->exit_code, &cmd->exit_why
, APR_WAIT);
    check(rv == APR_CHILD_DONE, "Failed to wait.");

    check(cmd->exit_code == 0, "%s exited badly.", cmd->exe);
    check(cmd->exit_why == APR_PROC_EXIT, "%s was killed or cras
hed", cmd->exe);

    return 0;

error:
    return -1;
}

Shell CLEANUP_SH = {
    .exe = "rm",
    .dir = "/tmp",
    .args = {"rm", "-rf", "/tmp/pkg-build", "/tmp/pkg-src.tar.gz"
,
        "/tmp/pkg-src.tar.bz2", "/tmp/DEPENDS", NULL}
};

Shell GIT_SH = {
    .dir = "/tmp",
    .exe = "git",
    .args = {"git", "clone", "URL", "pkg-build", NULL}
};

Shell TAR_SH = {
    .dir = "/tmp/pkg-build",
    .exe = "tar",
    .args = {"tar", "-xzf", "FILE", "--strip-components", "1", N
ULL}
};

Shell CURL_SH = {
    .dir = "/tmp",
    .exe = "curl",
    .args = {"curl", "-L", "-o", "TARGET", "URL", NULL}
};
```

```
Shell CONFIGURE_SH = {
    .exe = "./configure",
    .dir = "/tmp/pkg-build",
    .args = {"configure", "OPTS", NULL},
};

Shell MAKE_SH = {
    .exe = "make",
    .dir = "/tmp/pkg-build",
    .args = {"make", "OPTS", NULL}
};

Shell INSTALL_SH = {
    .exe = "sudo",
    .dir = "/tmp/pkg-build",
    .args = {"sudo", "make", "TARGET", NULL}
};
```

自底向上阅读 `shell.c` 的代码（这也是常见的C源码布局），你会看到我创建了实际的 `Shell` 变量，它在 `shell.h` 中以 `extern` 修饰。它们虽然在这里，但是也被程序的其它部分使用。这就是创建全局变量的方式，它们可以存在于一个 `.c` 文件中，但是可在任何地方使用。你不应该创建很多这类变量，但是它们的确很方便。

继续阅读代码，我们读到了 `Shell_run` ，它是一个“基”函数，只是基于 `Shell` 中的东西执行命令。它使用了许多在 `apr_thread_proc.h` 中定义的函数，你需要查阅它们的每一个来了解工作原理。这就像是一些使用 `system` 函数调用的代码一样，但是它可以让你控制其他程序的执行。例如，在我们的 `Shell` 结构中，存在 `.dir` 属性在运行之前强制程序必须在指定目录中。

最后，我创建了 `Shell_exec` 函数，它是个变参函数。你在之前已经看到过了，但是确保你理解了 `stdarg.h` 函数以及如何编写它们。在下个挑战中你需要分析这一函数。

挑战2：分析 `Shell_exec`

为这些文件（以及向挑战1那样的完整的代码复查）设置的挑战是完整分析 `Shell_exec` ，并且拆分代码来了解工作原理。你应该能够理解每一行代码， `for` 循环如何工作，以及参数如何被替换。

一旦你分析完成，向 `struct Shell` 添加一个字段，提供需要替代的 `args` 变量的数量。更新所有命令来接受参数的正确数量，随后增加一个错误检查，来确认参数被正确替换，以及在错误时退出。

命令行函数

现在你需要构造正确的命令来完成功能。这些命令会用到APR的函数、`db.h` 和 `shell.h` 来执行下载和构建软件的真正工作。这些文件最为复杂，所以要小心编写它们。你需要首先编写 `commands.h` 文件，接着在 `commands.c` 文件中实现它的函数。

```
#ifndef _commands_h
#define _commands_h

#include <apr_pools.h>

#define DEPENDS_PATH "/tmp/DEPENDS"
#define TAR_GZ_SRC "/tmp/pkg-src.tar.gz"
#define TAR_BZ2_SRC "/tmp/pkg-src.tar.bz2"
#define BUILD_DIR "/tmp/pkg-build"
#define GIT_PAT "*.git"
#define DEPEND_PAT "*DEPENDS"
#define TAR_GZ_PAT "*.tar.gz"
#define TAR_BZ2_PAT "*.tar.bz2"
#define CONFIG_SCRIPT "/tmp/pkg-build/configure"

enum CommandType {
    COMMAND_NONE, COMMAND_INSTALL, COMMAND_LIST, COMMAND_FETCH,
    COMMAND_INIT, COMMAND_BUILD
};


int Command_fetch(apr_pool_t *p, const char *url, int fetch_only)
;

int Command_install(apr_pool_t *p, const char *url, const char *
configure_opts,
        const char *make_opts, const char *install_opts);

int Command_depends(apr_pool_t *p, const char *path);

int Command_build(apr_pool_t *p, const char *url, const char *co
nfigure_opts,
        const char *make_opts, const char *install_opts);

#endif
```

`commands.h` 中并没有很多之前没见过的东西。你应该看到了一些字符串的定义，它们在任何地方都会用到。真正的代码在 `commands.c` 中。

```
#include <apr_uri.h>
#include <apr_fnmatch.h>
#include <unistd.h>

#include "commands.h"
```

```c
#include "dbg.h"
#include "bstrlib.h"
#include "db.h"
#include "shell.h"


int Command_depends(apr_pool_t *p, const char *path)
{
    FILE *in = NULL;
    bstring line = NULL;

    in = fopen(path, "r");
    check(in != NULL, "Failed to open downloaded depends: %s", p
ath);

    for(line = bgets((bNgetc)fgetc, in, '\n'); line != NULL;
            line = bgets((bNgetc)fgetc, in, '\n'))
    {
        btrimws(line);
        log_info("Processing depends: %s", bdata(line));
        int rc = Command_install(p, bdata(line), NULL, NULL, NULL
);
        check(rc == 0, "Failed to install: %s", bdata(line));
        bdestroy(line);
    }

    fclose(in);
    return 0;

error:
    if(line) bdestroy(line);
    if(in) fclose(in);
    return -1;
}

int Command_fetch(apr_pool_t *p, const char *url, int fetch_only)

{
    apr_uri_t info = {.port = 0};
    int rc = 0;
    const char *depends_file = NULL;
    apr_status_t rv = apr_uri_parse(p, url, &info);

    check(rv == APR_SUCCESS, "Failed to parse URL: %s", url);

    if(apr_fnmatch(GIT_PAT, info.path, 0) == APR_SUCCESS) {
        rc = Shell_exec(GIT_SH, "URL", url, NULL);
        check(rc == 0, "git failed.");
    } else if(apr_fnmatch(DEPEND_PAT, info.path, 0) == APR_SUCCE
SS) {
        check(!fetch_only, "No point in fetching a DEPENDS file."
);
```

```
        if(info.scheme) {
            depends_file = DEPENDS_PATH;
            rc = Shell_exec(CURL_SH, "URL", url, "TARGET", depen
ds_file, NULL);
            check(rc == 0, "Curl failed.");
        } else {
            depends_file = info.path;
        }

        // recursively process the devpkg list
        log_info("Building according to DEPENDS: %s", url);
        rv = Command_depends(p, depends_file);
        check(rv == 0, "Failed to process the DEPENDS: %s", url)
;

        // this indicates that nothing needs to be done
        return 0;

    } else if(apr_fnmatch(TAR_GZ_PAT, info.path, 0) == APR_SUCCE
SS) {
        if(info.scheme) {
            rc = Shell_exec(CURL_SH,
                    "URL", url,
                    "TARGET", TAR_GZ_SRC, NULL);
            check(rc == 0, "Failed to curl source: %s", url);
        }

        rv = apr_dir_make_recursive(BUILD_DIR,
                APR_UREAD | APR_UWRITE | APR_UEXECUTE, p);
        check(rv == APR_SUCCESS, "Failed to make directory %s",
BUILD_DIR);

        rc = Shell_exec(TAR_SH, "FILE", TAR_GZ_SRC, NULL);
        check(rc == 0, "Failed to untar %s", TAR_GZ_SRC);
    } else if(apr_fnmatch(TAR_BZ2_PAT, info.path, 0) == APR_SUCC
ESS) {
        if(info.scheme) {
            rc = Shell_exec(CURL_SH, "URL", url, "TARGET", TAR_B
Z2_SRC, NULL);
            check(rc == 0, "Curl failed.");
        }

        apr_status_t rc = apr_dir_make_recursive(BUILD_DIR,
                APR_UREAD | APR_UWRITE | APR_UEXECUTE, p);

        check(rc == 0, "Failed to make directory %s", BUILD_DIR)
;
        rc = Shell_exec(TAR_SH, "FILE", TAR_BZ2_SRC, NULL);
        check(rc == 0, "Failed to untar %s", TAR_BZ2_SRC);
    } else {
        sentinel("Don't now how to handle %s", url);
    }
```

```
    // indicates that an install needs to actually run
    return 1;
error:
    return -1;
}

int Command_build(apr_pool_t *p, const char *url, const char *co
nfigure_opts,
        const char *make_opts, const char *install_opts)
{
    int rc = 0;

    check(access(BUILD_DIR, X_OK | R_OK | W_OK) == 0,
            "Build directory doesn't exist: %s", BUILD_DIR);

    // actually do an install
    if(access(CONFIG_SCRIPT, X_OK) == 0) {
        log_info("Has a configure script, running it.");
        rc = Shell_exec(CONFIGURE_SH, "OPTS", configure_opts, NU
LL);
        check(rc == 0, "Failed to configure.");
    }

    rc = Shell_exec(MAKE_SH, "OPTS", make_opts, NULL);
    check(rc == 0, "Failed to build.");

    rc = Shell_exec(INSTALL_SH,
            "TARGET", install_opts ? install_opts : "install",
            NULL);
    check(rc == 0, "Failed to install.");

    rc = Shell_exec(CLEANUP_SH, NULL);
    check(rc == 0, "Failed to cleanup after build.");

    rc = DB_update(url);
    check(rc == 0, "Failed to add this package to the database."
);

    return 0;

error:
    return -1;
}

int Command_install(apr_pool_t *p, const char *url, const char *
configure_opts,
        const char *make_opts, const char *install_opts)
{
    int rc = 0;
    check(Shell_exec(CLEANUP_SH, NULL) == 0, "Failed to cleanup
before building.");

    rc = DB_find(url);
```

```
    check(rc != -1, "Error checking the install database.");

    if(rc == 1) {
        log_info("Package %s already installed.", url);
        return 0;
    }

    rc = Command_fetch(p, url, 0);

    if(rc == 1) {
        rc = Command_build(p, url, configure_opts, make_opts, in
stall_opts);
        check(rc == 0, "Failed to build: %s", url);
    } else if(rc == 0) {
        // no install needed
        log_info("Depends successfully installed: %s", url);
    } else {
        // had an error
        sentinel("Install failed: %s", url);
    }

    Shell_exec(CLEANUP_SH, NULL);
    return 0;

error:
    Shell_exec(CLEANUP_SH, NULL);
    return -1;
}
```

在你输入并编译它之后，就可以开始分析了。如果到目前为止你完成了前面的挑战，你会理解如何使用 `shell.c` 函数来运行shell命令，以及参数如何被替换。如果没有则需要回退到前面的挑战，确保你真正理解了 `Shell_exec` 的工作原理。

挑战3：评判我的设计

像之前一样，完整地复查一遍代码来保证一模一样。接着浏览每个函数并且确保你知道他如何工作。你也应该跟踪这个文件或其它文件中，每个函数对其它函数的调用。最后，确认你理解了这里的所有调用APR的函数。

一旦你正确编写并分析了这个文件，把我当成一个傻瓜一样来评判我的设计，我需要看看你是否可以改进它。不要真正修改代码，只是创建一个 `notes.txt` 并且写下你的想法和你需要修改的地方。

`devpkg` 的 `main` 函数

`devpkg.c` 是最后且最重要的，但是也可能是最简单的文件，其中创建了 `main` 函数。没有与之配套的 `.h` 文件，因为这个文件包含其他所有文件。这个文件用于创建 `devpkg` 可执行程序，同时组装了来自 `Makefile` 的其它 `.o` 文件。在文件中输入代码并保证正确。

```
#include <stdio.h>
#include <apr_general.h>
#include <apr_getopt.h>
#include <apr_strings.h>
#include <apr_lib.h>

#include "dbg.h"
#include "db.h"
#include "commands.h"

int main(int argc, const char const *argv[])
{
    apr_pool_t *p = NULL;
    apr_pool_initialize();
    apr_pool_create(&p, NULL);

    apr_getopt_t *opt;
    apr_status_t rv;

    char ch = '\0';
    const char *optarg = NULL;
    const char *config_opts = NULL;
    const char *install_opts = NULL;
    const char *make_opts = NULL;
    const char *url = NULL;
    enum CommandType request = COMMAND_NONE;


    rv = apr_getopt_init(&opt, p, argc, argv);

    while(apr_getopt(opt, "I:Lc:m:i:d:SF:B:", &ch, &optarg) == A
PR_SUCCESS) {
        switch (ch) {
            case 'I':
                request = COMMAND_INSTALL;
                url = optarg;
                break;

            case 'L':
                request = COMMAND_LIST;
                break;

            case 'c':
                config_opts = optarg;
                break;

            case 'm':
```

```
                make_opts = optarg;
                break;

            case 'i':
                install_opts = optarg;
                break;

            case 'S':
                request = COMMAND_INIT;
                break;

            case 'F':
                request = COMMAND_FETCH;
                url = optarg;
                break;

            case 'B':
                request = COMMAND_BUILD;
                url = optarg;
                break;
        }
    }

    switch(request) {
        case COMMAND_INSTALL:
            check(url, "You must at least give a URL.");
            Command_install(p, url, config_opts, make_opts, inst
all_opts);
            break;

        case COMMAND_LIST:
            DB_list();
            break;

        case COMMAND_FETCH:
            check(url != NULL, "You must give a URL.");
            Command_fetch(p, url, 1);
            log_info("Downloaded to %s and in /tmp/", BUILD_DIR)
;
            break;

        case COMMAND_BUILD:
            check(url, "You must at least give a URL.");
            Command_build(p, url, config_opts, make_opts, instal
l_opts);
            break;

        case COMMAND_INIT:
            rv = DB_init();
            check(rv == 0, "Failed to make the database.");
            break;

        default:
```

```
            sentinel("Invalid command given.");
    }


    return 0;

error:
    return 1;
}
```

挑战4：README 和测试文件

为这个文件设置的挑战是理解参数如何处理，以及参数是什么，之后创建含有使用指南的 `README` 文件。在编写 `README` 的同时，也编写一个简单的 `simple.sh` ，它运行 `./devpkg` 来检查每个命令都在实际环境下工作。在你的脚本顶端使用 `set -e` ，使它跳过第一个错误。

最后，在 `Valgrind` 下运行程序，确保在进行下一步之前，所有东西都能正常运行。

期中检测

最后的挑战就是这个期中检测，它包含三件事情：

- 将你的代码与我的在线代码对比，以100%的分数开始，每错一行减去1%。
- 在你的 `notes.txt` 中记录你是如何改进代码和 `devpkg` 的功能，并且实现你的改进。
- 编写一个 `devpkg` 的替代版本，使用其他你喜欢的语言，或者你觉得最适合编写它的语言。对比二者，之后基于你的结果改进你的 `devpkg` 的C版本。

你可以执行下列命令来将你的代码与我的对比：

```
cd ..  # get one directory above your current one
git clone git://gitorious.org/devpkg/devpkg.git devpkgzed
diff -r devpkg devpkgzed
```

这将会克隆我的 `devpkg` 版本到 `devpkgzed` 目录中。之后使用工具 `diff` 来对比你的和我的代码。书中你所使用的这些文件直接来自于这个项目，所以如果出现了不同的行，肯定就有错误。

要记住这个练习没有真正的及格或不及格，它只是一个方式来让你挑战自己，并尽可能变得精确和谨慎。

练习27：创造性和防御性编程

> 原文：Exercise 27: Creative And Defensive Programming
>
> 译者：飞龙

你已经学到了大多数C语言的基础，并且准备好开始成为一个更严谨的程序员了。这里就是从初学者走向专家的地方，不仅仅对于C，更对于核心的计算机科学概念。我将会教给你一些核心的数据结构和算法，它们是每个程序员都要懂的，还有一些我在真实程序中所使用的一些非常有趣的东西。

在我开始之前，我需要教给你一些基本的技巧和观念，它们能帮助你编写更好的软件。练习27到31会教给你高级的概念和特性，而不是谈论编程，但是这些之后你将会应用它们来编写核心库或有用的数据结构。

编写更好的C代码（实际上是所有语言）的第一步是，学习一种新的观念叫做“防御性编程”。防御性编程假设你可能会制造出很多错误，之后尝试在每一步尽可能预防它们。这个练习中我打算教给你如何以防御性的思维来思考编程。

创造性编程思维

在这个简单的练习中要告诉你如何做到创造性是不可能的，但是我会告诉你一些涉及到任务风险和开放思维的创造力。恐惧会快速地扼杀创造力，所以我采用，并且许多程序员也采用的这种思维方式使我不会惧怕风险，并且看上去像个傻瓜。

- 我不会犯错误。
- 人们所想的并不重要。
- 我脑子里面诞生的想法才是最好的。

我只是暂时接受了这种思维，并且在应用中用了一些小技巧。为了这样做我会提出一些想法，寻找创造性的解决方案，开一些奇奇怪怪的脑洞，并且不会害怕发明一些古怪的东西。在这种思维方式下，我通常会编写出第一个版本的糟糕代码，用于将想法描述出来。

然而，当我完成我的创造性原型时，我会将它扔掉，并且将它变得严谨和可考。其它人在这里常犯的一个错误就是将创造性思维引入它们的实现阶段。这样会产生一种非常不同的破坏性思维，它是创造性思维的阴暗面：

- 编写完美的软件是可行的。
- 我的大脑告诉我了真相，它不会发现任何错误，所以我写了完美的软件。
- 我的代码就是我自己，批判它的人也在批判我。

这些都是错误的。你经常会碰到一些程序员，它们对自己创造的软件具有强烈的荣誉感。这很正常，但是这种荣誉感会成为客观上改进作品的阻力。由于这种荣誉感和它们对作品的依恋，它们会一直相信它们编写的东西是完美的。只要它们忽视其它人的对这些代码的观点，它们就可以保护它们的玻璃心，并且永远不会改进。

同时具有创造性思维和编写可靠软件的技巧是，采用防御性编程的思维。

防御性编程思维

在你做出创造性原型，并且对你的想法感觉良好之后，就应该切换到防御性思维了。防御性思维的程序员大致上会否定你的代码，并且相信下面这些事情：

- 软件中存在错误。
- 你并不是你的软件，但你需要为错误负责。
- 你永远不可能消除所有错误，只能降低它们的可能性。

这种思维方式让你诚实地对待你的代码，并且为改进批判地分析它。注意上面并没有说你充满了错误，只是说你的代码充满错误。这是一个需要理解的关键，因为它给了你编写下一个实现的客观力量。

就像创造性思维，防御性编程思维也有阴暗面。防御性程序员是一个惧怕任何事情的偏执狂，这种恐惧使他们远离可能的错误或避免犯错误。当你尝试做到严格一致或正确时这很好，但是它是创造力和专注的杀手。

八个防御性编程策略

一旦你接受了这一思维，你可以重新编写你的原型，并且遵循下面的八个策略，它们被我用于尽可能把代码变得可靠。当我编写代码的“实际”版本，我会严格按照下面的策略，并且尝试消除尽可能多的错误，以一些会破坏我软件的人的方式思考。

永远不要信任输入

永远不要提供的输入，并总是校验它。

避免错误

如果错误可能发生，不管可能性多低都要避免它。

过早暴露错误

过早暴露错误，并且评估发生了什么、在哪里发生以及如何修复。

记录假设

清楚地记录所有先决条件，后置条件以及不变量。

防止过多的文档

不要在实现阶段就编写文档，它们可以在代码完成时编写。

使一切自动化

使一切自动化，尤其是测试。

简单化和清晰化

永远简化你的代码，在没有牺牲安全性的同时变得最小和最整洁。

质疑权威

不要盲目遵循或拒绝规则。

这些并不是全部，仅仅是一些核心的东西，我认为程序员应该在编程可靠的代码时专注于它们。要注意我并没有真正说明如何具体做到这些，我接下来会更细致地讲解每一条，并且会布置一些覆盖它们的练习。

应用这八条策略

这些观点都是一些流行心理学的陈词滥调，但是你如何把它们应用到实际编程中呢？我现在打算向你展示这本书中的一些代码所做的事情，这些代码用具体的例子展示每一条策略。这八条策略并不止于这些例子，你应该使用它们作为指导，使你的代码更可靠。

永远不要信任输入

让我们来看一个坏设计和“更好”的设计的例子。我并不想称之为好设计，因为它可以做得更好。看一看这两个函数，它们都复制字符串， `main` 函数用于测试哪个更好。

```
undef NDEBUG
#include "dbg.h"
#include <stdio.h>
#include <assert.h>

/*
 * Naive copy that assumes all inputs are always valid
 * taken from K&R C and cleaned up a bit.
 */
void copy(char to[], char from[])
{
    int i = 0;

    // while loop will not end if from isn't '\0' terminated
    while((to[i] = from[i]) != '\0') {
        ++i;
    }
}

/*
 * A safer version that checks for many common errors using the
 * length of each string to control the loops and termination.
 */
int safercopy(int from_len, char *from, int to_len, char *to)
{
    assert(from != NULL && to != NULL && "from and to can't be N
```

```
ULL");
    int i = 0;
    int max = from_len > to_len - 1 ? to_len - 1 : from_len;

    // to_len must have at least 1 byte
    if(from_len < 0 || to_len <= 0) return -1;

    for(i = 0; i < max; i++) {
        to[i] = from[i];
    }

    to[to_len - 1] = '\0';

    return i;
}


int main(int argc, char *argv[])
{
    // careful to understand why we can get these sizes
    char from[] = "0123456789";
    int from_len = sizeof(from);

    // notice that it's 7 chars + \0
    char to[] = "0123456";
    int to_len = sizeof(to);

    debug("Copying '%s':%d to '%s':%d", from, from_len, to, to_l
en);

    int rc = safercopy(from_len, from, to_len, to);
    check(rc > 0, "Failed to safercopy.");
    check(to[to_len - 1] == '\0', "String not terminated.");

    debug("Result is: '%s':%d", to, to_len);

    // now try to break it
    rc = safercopy(from_len * -1, from, to_len, to);
    check(rc == -1, "safercopy should fail #1");
    check(to[to_len - 1] == '\0', "String not terminated.");

    rc = safercopy(from_len, from, 0, to);
    check(rc == -1, "safercopy should fail #2");
    check(to[to_len - 1] == '\0', "String not terminated.");

    return 0;

error:
    return 1;
}
```

`copy` 函数是典型的C代码，而且它是大量缓冲区溢出的来源。它有缺陷，因为它总是假设接受到的是合法的C字符串（带有 `'\0'` ），并且只是用一个 `while` 循环来处理。问题是，确保这些是十分困难的，并且如果没有处理好，它会使 `while` 循环无限执行。编写可靠代码的一个要点就是，不要编写可能不会终止的循环。

`safecopy` 函数尝试通过要求调用者提供两个字符串的长度来解决问题。它可以执行有关这些字符串的、 `copy` 函数不具备的特定检查。他可以保证长度正确， `to` 字符串具有足够的容量，以及它总是可终止。这个函数不像 `copy` 函数那样可能会永远执行下去。

这个就是永远不信任输入的实例。如果你假设你的函数要接受一个没有终止标识的字符串（通常是这样），你需要设计你的函数，不要依赖字符串本身。如果你想让参数不为 `NULL` ，你应该对此做检查。如果大小应该在正常范围内，也要对它做检查。你只需要简单假设调用你代码的人会把它弄错，并且使他们更难破坏你的函数。

这个可以扩展到从外部环境获取输入的的软件。程序员著名的临终遗言是，“没人会这样做。”我看到他们说了这句话后，第二天有人就这样做，黑掉或崩溃它们的应用。如果你说没有人会这样做，那就加固代码来保证他们不会简单地黑掉你的应用。你会因所做的事情而感到高兴。

这种行为会出现收益递减。下面是一个清单，我会尝试对我用C写的每个函数做如下工作：

- 对于每一个参数定义它的先决条件，以及这个条件是否导致失效或返回错误值。如果你在编写一个库，比起失效要更倾向于错误。
- 对于每个先决条件，使用 `assert(test && "message");` 在最开始添加 `assert` 检查。这句代码会执行检查，失败时OS通常会打印断言行，通常它包括信息。当你尝试弄清 `assert` 为什么在这里时，这会非常有用。
- 对于其它先决条件，返回错误代码或者使用我的 `check` 宏来执行它并且提供错误信息。我在这个例子中没有使用 `check` ，因为它会混淆比较。
- 记录为什么存在这些先决条件，当一个程序员碰到错误时，他可以弄清楚这些是否是真正必要的。
- 如果你修改了输入，确保当函数退出或中止时它们也会正确产生。
- 总是要检查所使用的函数的错误代码。例如，人们有时会忘记检查 `fopen` 或 `fread` 的返回代码，这会导致他们在错误下仍然使用这个资源。这会导致你的程序崩溃或者易受攻击。
- 你也需要返回一致的错误代码，以便对你的每个函数添加相同的机制。一旦你熟悉了这一习惯，你就会明白为什么我的 `check` 宏这样工作。

只是这些微小的事情就会改进你的资源处理方式，并且避免一大堆错误。

避免错误

上一个例子中你可能会听到别人说，“程序员不会经常错误地使用 `copy` 。”尽管大量攻击都针对这类函数，他们仍旧相信这种错误的概率非常低。概率是个很有趣的事情，因为人们不擅长猜测所有事情的概率，这非常难以置信。然而人们对于判断

一个事情是否可能，是很擅长的。他们可能会说 `copy` 中的错误不常见，但是无法否认它可能发生。

关键的原因是对于一些常见的事情，它首先是可能的。判断可能性非常简单，因为我们都知道事情如何发生。但是随后判断出概率就不是那么容易了。人们错误使用 `copy` 的情况会占到20%、10%，或1%？没有人知道。为了弄清楚你需要收集证据，统计许多软件包中的错误率，并且可能需要调查真实的程序员如何使用这个函数。

这意味着，如果你打算避免错误，你不需要尝试避免可能发生的事情，而是要首先集中解决概率最大的事情。解决软件所有可能崩溃的方式并不可行，但是你可以尝试一下。同时，如果你不以最少的努力解决最可能发生的事件，你就是在不相关的风险上浪费时间。

下面是一个决定避免什么的处理过程：

- 列出所有可能发生的错误，无论概率大小，并带着它们的原因。不要列出外星人可能会监听内存来偷走密码这样的事情。
- 评估每个的概率，使用危险行为的百分比来表示。如果你处理来自互联网的情况，那么则为可能出现错误的请求的百分比。如果是函数调用，那么它是出现错误的函数调用百分比。
- 评估每个的工作量，使用避免它所需的代码量或工作时长来表示。你也可以简单给它一个“容易”或者“难”的度量。当需要修复的简单错误仍在列表上时，任何这种度量都可以让你避免做无谓的工作。
- 按照工作量（低到高）和概率（高到低）排序，这就是你的任务列表。
- 之后避免你在列表中列出的任何错误，如果你不能消除它的可能性，要降低它的概率。
- 如果存在你不能修复的错误，记录下来并提供给可以修复的人。

这一微小的过程会产生一份不错的待办列表。更重要的是，当有其它重要的事情需要解决时，它让你远离劳而无功。你也可以更正式或更不正式地处理这一过程。如果你要完成整个安全审计，你最好和团队一起做，并且有个更详细的电子表格。如果你只是编写一个函数，简单地复查代码之后划掉它们就够了。最重要的是你要停止假设错误不会发生，并且着力于消除它们，这样就不会浪费时间。

过早暴露错误

如果你遇到C中的错误，你有两个选择：

- 返回错误代码。
- 中止进程。

这就是处理方法，你需要执行它来确保错误尽快发生，记录清楚，提供错误信息，并且易于程序员来避免它。这就是我提供的 `check` 宏这样工作的原因。对于每一个错误，你都要让它你打印信息、文件名和行号，并且强制返回错误代码。如果你使用了我的宏，你会以正确的方式做任何事情。

我倾向于返回错误代码而不是终止程序。如果出现了大错误我会中止程序，但是实际上我很少碰到大错误。一个需要中止程序的很好例子是，我获取到了一个无效的指针，就像 `safecopy` 中那样。我没有让程序在某个地方产生“段错误”，而是立即

捕获并中止。但是，如果传入`NULL`十分普遍，我可能会改变方式而使用`check`来检查，以保证调用者可以继续运行。

然而在库中，我尽我最大努力永不中止。使用我的库的软件可以决定是否应该中止。如果这个库使用非常不当，我才会中止程序。

最后，关于“暴露”的一大部分内容是，不要对多于一个错误使用相同的信息或错误代码。你通常会在外部资源的错误中见到这种情况。比如一个库捕获了套接字上的错误，之后简单报告“套接字错误”。它应该做的是返回具体的信息，比如套接字上发生了什么错误，使它可以被合理地调试和修复。当你设计错误报告时，确保对于不同的错误你提供了不同的错误消息。

记录假设

如果你遵循并执行了这个建议，你就构建了一份“契约”，关于函数期望这个世界是什么样子。你已经为每个参数预设了条件，处理潜在的错误，并且优雅地产生失败。下一步是完善这一契约，并且添加“不变量”和“后置条件”。

不变量就是在函数运行时，一些场合下必须恒为真的条件。这对于简单的函数并不常见，但是当你处理复杂的结构时，它会变得很必要。一个关于不变量的很好的例子是，结构体在使用时都会合理地初始化。另一个是有序的数据结构在处理时总是排好序的。

后置条件就是退出值或者函数运行结果的保证。这可以和不变了混在一起，但是也可以是一些很简单的事情，比如“函数应总是返回0，或者错误时返回-1”。通常这些都有文档记录，但是如果你的函数返回一个分配的资源，你应该添加一个后置条件，做检查来确保它返回了一个不为`NULL`的东西。或者，你可以使用`NULL`来表示错误，这种情况下，你的后置条件就是资源在任何错误时都会被释放。

在C编程中，不变量和后置条件都通常比实际的代码和断言更加文档化。处理它们的最好当时就是尽可能添加`assert`调用，之后记录剩下的部分。如果你这么做了，当其它人碰到错误时，他们可以看到你在编写函数时做了什么假设。

避免过多文档

程序员编写代码时的一个普遍问题，就是他们会记录一个普遍的bug，而不是简单地修复它。我最喜欢的方式是，Ruby on Rails系统只是简单地假设所有月份都有30天。日历太麻烦了，所以与其修复它，不如在一些地方放置一个小的注释，说这是故意的，并且几年内都不会改正。每次一些人试图抱怨它时，他们都会说，“文档里面都有！”

如果你能够实际修复问题，文档并不重要，并且，如果函数具有严重的缺陷，你在修复它之前可以不记录它。在Ruby on Rails的例子中，不包含日期函数会更好一些，而不是包含一个没人会用的错误的函数。

当你为防御性编程执行清理时，尽可能尝试修复任何事情。如果你发现你记录了越来越多的，你不能修复的事情，需要考虑重新设计特性，或简单地移除它。如果你真的需要保留这一可怕的错误的特性，那么我建议你编写它、记录它，并且在你受责备之前找一份新的工作。

使一切自动化

你是个程序员，这意味着你的工作是通过自动化消灭其它人的工作。它的终极目标是使用自动化来使你自己也失业。很显然你不应该完全消除你做的东西，但如果你花了一整天在终端上重复运行手动测试，你的工作就不是编程。你只是在做QA，并且你应该使自己自动化，消除这个你可能并不是真的想干的QA工作。

实现它的最简单方式就是编写自动化测试，或者单元测试。这本书里我打算讲解如何使它更简单，并且我会避免多数编写测试的信条。我只会专注于如何编写它们，测试什么，以及如何使测试更高效。

下面是程序员没有但是应该自动化的一些事情：

- 测试和校验。
- 构建过程。
- 软件部署。
- 系统管理。
- 错误报告。

尝试花一些时间在自动化上面，你会有更多的时间用来处理一些有趣的事情。或者，如果这对你来说很有趣，也许你应该编写自动化完成这些事情的软件。

简单化和清晰化

“简单性”的概念对许多人来说比较微妙，尤其是一些聪明人。它们通常将“内涵”与“简单性”混淆起来。如果他们很好地理解了它，很显然非常简单。简单性的测试是通过将一个东西与比它更简单的东西比较。但是，你会看到编写代码的人会使用最复杂的、匪夷所思的数据结构，因为它们认为做同样事情的简单版本非常“恶心”。对复杂性的爱好是程序员的弱点。

你可以首先通过告诉自己，“简单和清晰并不恶心，无论谁在干什么事情”来战胜这一弱点。如果其它人编写了愚蠢的观察者模式涉及到19个类，12个接口，而你只用了两个字符串操作就可以实现它，那么你赢了。他们就是错了，无论他们认为自己的复杂设计有多么高大上。

对于要使用哪个函数的最简单测试是：

- 确保所有函数都没有问题。如果它有错误，它有多快或多简单就不重要了。
- 如果你不能修复问题，就选择另外一个。
- 它们会产生相同结果嘛？如果不是就挑选具有所需结果的函数。
- 如果它们会产生相同结果，挑选包含更少特性，更少分支的那个，或者挑选你认为最简单的那个。
- 确保你没有只是挑选最具有表现力的那个。无论怎么样，简单和清晰，都会战胜复杂和恶心。

你会注意到，最后我一般会放弃并告诉你根据你的判断。简单性非常讽刺地是一件复杂的事情，所以使用你的品位作为指引是最好的方式。只需要确保在你获取更多经验之后，你会调整你对于什么是“好”的看法。

质疑权威

最后一个策略是最重要的，因为它让你突破防御性编程思维，并且让你转换为创造性思维。防御性编程是权威性的，并且比较无情。这一思维方式的任务是让你遵循规则，因为否则你会错失一些东西或心烦意乱。

这一权威性的观点的坏处是扼杀了独立的创造性思维。规则对于完成事情是必要的，但是做它们的奴隶会扼杀你的创造力。

这条最后的策略的意思是你应该周期性地质疑你遵循的规则，并且假设它们都是错误的，就像你之前复查的软件那样。在一段防御性编程的时间之后，我通常会这样做，我会拥有一个不编程的休息并让这些规则消失。之后我会准备好去做一些创造性的工作，或按需做更多的防御型编程。

顺序并不重要

在这一哲学上我想说的最后一件事，就是我并不是告诉你要按照一个严格的规则，比如“创造！防御！创造！防御！”去做这件事。最开始你可能想这样做，但是我实际上会做不等量的这些事情，取决于我想做什么，并且我可能会将二者融合到一起，没有明确的边界。

我也不认为其中一种思维会优于另一种，或者它们之间有严格的界限。你需要在编程上既有创造力也要严格，所以如果想要提升的话，需要同时做到它们。

附加题

- 到现在为止（以及以后）书中的代码都可能违反这些规则。回退并挑选一个练习，将你学到的应用在它上面，来看看你能不能改进它或发现bug。
- 寻找一个开源项目，对其中一些文件进行类似的代码复查。如果你发现了bug，提交一个补丁来修复它。

练习28：Makefile 进阶

> 原文：Exercise 28: Intermediate Makefiles
>
> 译者：飞龙

在下面的三个练习中你会创建一个项目的目录框架，用于构建之后的C程序。这个目录框架会在这本书中剩余的章节中使用，并且这个练习中我会涉及到 `Makefile` 便于你理解它。

这个结构的目的是，在不凭借配置工具的情况下，使构建中等规模的程序变得容易。如果完成了它，你会学到很多GNU make和一些小型shell脚本方面的东西。

基本的项目结构

首先要做的事情是创建一个C的目录框架，并且放置一些多续项目都拥有的，基本的文件和目录。这是我的目录：

```
$ mkdir c-skeleton
$ cd c-skeleton/
$ touch LICENSE README.md Makefile
$ mkdir bin src tests
$ cp dbg.h src/   # this is from Ex20
$ ls -l
total 8
-rw-r--r--  1 zedshaw  staff     0 Mar 31 16:38 LICENSE
-rw-r--r--  1 zedshaw  staff  1168 Apr  1 17:00 Makefile
-rw-r--r--  1 zedshaw  staff     0 Mar 31 16:38 README.md
drwxr-xr-x  2 zedshaw  staff    68 Mar 31 16:38 bin
drwxr-xr-x  2 zedshaw  staff    68 Apr  1 10:07 build
drwxr-xr-x  3 zedshaw  staff   102 Apr  3 16:28 src
drwxr-xr-x  2 zedshaw  staff    68 Mar 31 16:38 tests
$ ls -l src
total 8
-rw-r--r--  1 zedshaw  staff  982 Apr  3 16:28 dbg.h
$
```

之后你会看到我执行了 `ls -l` ，所以你会看到最终结果。

下面是每个文件所做的事情：

`LICENSE`

如果你在项目中发布源码，你会希望包含一份协议。如果你不这么多，虽然你有代码的版权，但是通常没有人有权使用。

`README.md`

对你项目的简要说明。它以`.md`结尾，所以应该作为Markdown来解析。

`Makefile`

这个项目的主要构建文件。

`bin/`

放置可运行程序的地方。这里通常是空的，Makefile会在这里生成程序。

`build/`

当值库和其它构建组件的地方。通常也是空的，Makefile会在这里生成这些东西。

`src/`

放置源码的地方，通常是`.c`和`.h`文件。

`tests/`

放置自动化测试的地方。

`src/dbg.h`

我将练习20的`dbg.h`复制到了这里。

我刚才分解了这个项目框架的每个组件，所以你应该明白它们怎么工作。

Makefile

我要讲到的第一件事情就是Makefile，因为你可以从中了解其它东西的情况。这个练习的Makeile比之前更加详细，所以我会在你输入它之后做详细的分解。

```
CFLAGS=-g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG $(OPTFLAGS)
LIBS=-ldl $(OPTLIBS)
PREFIX?=/usr/local

SOURCES=$(wildcard src/**/*.c src/*.c)
OBJECTS=$(patsubst %.c,%.o,$(SOURCES))

TEST_SRC=$(wildcard tests/*_tests.c)
TESTS=$(patsubst %.c,%,$(TEST_SRC))

TARGET=build/libYOUR_LIBRARY.a
SO_TARGET=$(patsubst %.a,%.so,$(TARGET))

# The Target Build
all: $(TARGET) $(SO_TARGET) tests

dev: CFLAGS=-g -Wall -Isrc -Wall -Wextra $(OPTFLAGS)
dev: all

$(TARGET): CFLAGS += -fPIC
```

```
$(TARGET): build $(OBJECTS)
        ar rcs $@ $(OBJECTS)
        ranlib $@

$(SO_TARGET): $(TARGET) $(OBJECTS)
        $(CC) -shared -o $@ $(OBJECTS)

build:
        @mkdir -p build
        @mkdir -p bin

# The Unit Tests
.PHONY: tests
tests: CFLAGS += $(TARGET)
tests: $(TESTS)
        sh ./tests/runtests.sh

valgrind:
        VALGRIND="valgrind --log-file=/tmp/valgrind-%p.log" $(MAK
E)

# The Cleaner
clean:
        rm -rf build $(OBJECTS) $(TESTS)
        rm -f tests/tests.log
        find . -name "*.gc*" -exec rm {} \;
        rm -rf `find . -name "*.dSYM" -print`

# The Install
install: all
        install -d $(DESTDIR)/$(PREFIX)/lib/
        install $(TARGET) $(DESTDIR)/$(PREFIX)/lib/

# The Checker
BADFUNCS='[^_.>a-zA-Z0-9](str(n?cpy|n?cat|xfrm|n?dup|str|pbrk|to
k|_)|stpn?cpy|a?sn?printf|byte_)'
check:
        @echo Files with potentially dangerous functions.
        @egrep $(BADFUNCS) $(SOURCES) || true
```

要记住你应该使用一致的Tab字符来缩进Makefile。你的编辑器应该知道怎么做，但是如果不是这样你可以换个编辑器。没有程序员会使用一个连如此简单的事情都做不好的编辑器。

头部

这个Makefile设计用于构建一个库，我们之后会用到它，并且通过使用 `GNU make` 的特殊特性使它在任何平台上都可用。我会在这一节拆分它的每一部分，先从头部开始。

Makefile:1

这是通常的 `CFLAGS` ，几乎每个项目都会设置，但是带有用于构建库的其它东西。你可能需要为不同平台调整它。要注意最后的 `OPTFLAGS` 变量可以让使用者按需扩展构建选项。

Makefile:2

用于链接库的选项，同样也允许其它人使用 `OPTFLAGS` 变量扩展链接选项。

Makefile:3

设置一个叫做 `PREFIX` 的可选变量，它只在没有 `PREFIX` 设置的平台上运行Makefile时有效。这就是 `?=` 的作用。

Makefile:5

这神奇的一行通过执行 `wildcard` 搜索在 `src/` 中所有 `*.c` 文件来动态创建 `SOURCES` 变量。你需要提供 `src/**/*.c` 和 `src/*.c` 以便GNU make能够包含 `src` 目录及其子目录的所有此类文件。

Makefile:6

一旦你创建了源文件列表，你可以使用 `patsubst` 命令获取 `*.c` 文件的 `SOURCES` 来创建目标文件的新列表。你可以告诉 `patsubst` 把所有 `%.c` 扩展为 `%.o` ，并将它们赋给 `OBJECTS` 。

Makefile:8

再次使用 `wildcard` 来寻找所有用于单元测试的测试源文件。它们存放在不同的目录中。

Makefile:9

之后使用相同的 `patsubst` 技巧来动态获得所有 `TEST` 目标。其中我去掉了 `.c` 后缀，使整个程序使用相同的名字创建。之前我将 `.c` 替换为 `.o` 来创建目标文件。

Makefile:11

最后，我将最终目标设置为 `build/libYOUR_LIBRARY.a` ，你可以为你实际构建的任何库来修改它。

这就是Makefile的头部了，但是我应该对“让其他人扩展构建”做个解释。你在运行它的时候可以这样做：

```
# WARNING! Just a demonstration, won't really work right now.
# this installs the library into /tmp
$ make PREFIX=/tmp install
# this tells it to add pthreads
$ make OPTFLAGS=-pthread
```

如果你传入匹配 `Makefile` 中相同名称的变量，它们会在构建中生效。你可以利用它来修改 `Makefile` 的运行方式。第一条命令改变了 `PREFIX` ，使它安装到 `/tmp` 。第二条设置了 `OPTFLAGS` ，为之添加了 `pthread` 选项。

构建目标

我会继续 `Makefile` 的分解，这一部分用于构建目标文件（object file）和目标（target）：

Makefile:14

要记住在没有提供目标时 `make` 会默认运行第一个目标。这里它叫做 `all:` ，并且它提供了 `$(TARGET) tests` 作为构建目标。查看 `TARGET` 变量，你会发现这就是库文件，所以 `all:` 首先会构建出库文件。之后， `tests` 目标会构建单元测试。

Makefile:16

另一个用于执行“开发者构建”的目标，它介绍了一种为单一目标修改选项的技巧，如果我执行“开发构建”，我希望 `CFLAGS` 包含类似 `Wextra` 这样用于发现bug的选项。如果你将它们放到目标的那行中，并再编写一行来指向原始目标（这里是 `all` ），那么它就会将改为你设置的选项。我通常将它用于在不同的平台上设置所需的不同选项。

Makefile:19

构建 `TARGET` 库，然而它同样使用了15行的技巧，向一个目标提供选项来为当前目标修改它们。这里我通过适用 `+=` 语法为库的构建添加了 `-fPIC` 。

Makefile:20

现在这一真实目标首先创建 `build` 目录，之后编译所有 `OBJECTS` 。

Makefile:21

运行实际创建 `TARGET` 的 `ar` 的命令。 `$@ $(OBJECTS)` 语法的意思是，将当前目标的名称放在这里，并把 `OBJECTS` 的内容放在后面。这里 `$@` 的值为19行的 `$(TARGET)` ，它实际上为 `build/libYOUR_LIBRARY.a` 。看起来在这一重定向中它做了很多跟踪工作，它也有这个功能，并且你可以通过修改顶部的 `TARGET` ，来构建一个全新的库。

Makefile:22

最后，在 `TARGET` 上运行 `ranlib` 来构建这个库。

Makefile:24-24

用于在 `build/` 和 `bin/` 目录不存在的条件下创建它们。之后它被19行引用，那里提供了 `build` 目标来确保 `build/` 目录已创建。

你现在拥有了用于构建软件的所需的所有东西。之后我们会创建用于构建和运行单元测试的东西，来执行自动化测试。

单元测试

C不同于其他语言，因为它更易于为每个需要测试的东西创建小型程序。一些测试框架试图模拟其他语言中的模块概念，并且执行动态加载，但是它在C中并不适用。这也不是必要的，因为你可以仅仅编写一个程序用于每个测试。

我接下来会涉及到Makefile的这一部分，并且你会看到 `test/` 目录中真正起作用的内容。

Makefile:29

如果你拥有一个不是“真实”的目标，只有有个目录或者文件叫这个名字，你需要使用g `.PHONY:` 标签来标记它，以便 `make` 忽略该文件。

Makefile:30

我使用了与修改 `CFLAGS` 变量相同的技巧，并且将 `TARGET` 添加到构建中，于是每个测试程序都会链接 `TARGET` 库。这里它会添加 `build/libYOUR_LIBRARY.a` 用于链接。

Makefile:31

之后我创建了实际的 `test:` 目录，它依赖于所有在 `TESTS` 变量中列出的程序。这一行实际上说，“Make，请使用你已知的程序构建方法，以及当前 `CFLAGS` 设置的内容来构建 `TESTS` 中的每个程序。”

Makefile:32

最后，所有 `TESTS` 构建完之后，会运行一个我稍后创建的简单shell脚本，它知道如何全部运行他们并报告它们的输出、这一行实际上运行它来让你看到测试结果。

Makefile:34-35

为了能够动态使用 `Valgrind` 重复运行测试，我创建了 `valgrind:` 标签，它设置了正确的变量并且再次运行它。它会将 `Valgrind` 的日志放到 `/tmp/valgrind-*.log` ，你可以查看并了解发生了什么。之后 `tests/runtests.sh` 看到 `VALGRIND` 变量时，它会明白要在 `Valgrind` 下运行测试程序。

你需要为单元测试创建一个小型的shell脚本，它知道如何运行程序。我们开始创建这个 `tests/runtests.sh` 脚本：

```
echo "Running unit tests:"

for i in tests/*_tests
do
    if test -f $i
    then
        if $VALGRIND ./$i 2>> tests/tests.log
        then
            echo $i PASS
        else
            echo "ERROR in test $i: here's tests/tests.log"
            echo "------"
            tail tests/tests.log
            exit 1
        fi
    fi
done

echo ""
```

当我提到单元测试如何工作时，我会在之后用到它。

清理工具

我已经有了用于单元测试的工具，所以下一步就是创建需要重置时的清理工具。

Makefile:38

`clean:` 目标在我需要清理这个项目的任何时候都会执行清理。

Makefile:39-42

这会清理不同编译器和工具留下的多数垃圾。它也会移除 `build/` 目录并且使用了一个技巧来清理XCode为调试目的而留下的 `*.dSYM` 。

如果你碰到了想要执行清理的垃圾，你只需要简单地扩展需要删除的文件列表。

安装

然后，我会需要一种安装项目的方法，对 `Makefile` 来说就是把构建出来的库放到通常的 `PREFIX` 目录下，它通常是 `/usr/local/lib` 。

Makefile:45

它会使 `install:` 依赖于 `all:` 目录，所以当你运行 `make install` 之后也会先确保一切都已构建。

Makefile:46

接下来我使用`install`程序来创建`lib`目标的目录。其中我通过使用两个为安装者提供便利的变量，尝试让安装尽可能灵活。`DESTDIR`交给安装者，便于在安全或者特定的目录里执行自己的构建。`PREFIX`在别人想要将项目安装到其它目录而不是`/user/local`时会被使用。

Makefile:47

在此之后我使用`insyall`来实际安装这个库，到它需要安装的地方。

`install`程序的目的是确保这些事情都设置了正确的权限。当你运行`make install`时你通常使用root权限来执行，所以通常的构建过程应为`make && sudo make install`。

检查工具

`Makefile`的最后一部分是个额外的部分，我把它包含在我的C项目中用于发现任何使用C中“危险”函数的情况。这些函数是字符串函数和另一些“不保护栈”的函数。

Makefile:50

设置变量，它是个稍大的正则表达式，用于检索类似`strcpy`的危险函数。

Makefile:51

这是`check:`目标，使你能够随时执行检查。

Makefile:52

它只是一个打印信息的方式，使用了`@echo`来告诉`make`不要打印命令，只需打印输出。

Makefile:53

对源文件运行`egrep`命令来寻找任何危险的字符串。最后的`|| true`是一种方法，用于防止`make`认为`egrep`没有找到任何东西是执行失败。

当你执行它之后，它会表现得十分奇怪，如果没有任何危险的函数，你会得到一个错误。

你会看到什么

我在完成这个项目框架目录的构建之前，还设置了两个额外的练习。下面这是我对`Makefile`特性的测试结果：

```
$ make clean
rm -rf build
rm -f tests/tests.log
find . -name "*.gc*" -exec rm {} \;
rm -rf `find . -name "*.dSYM" -print`
$ make check
Files with potentially dangerous functions.
^Cmake: *** [check] Interrupt: 2

$ make
ar rcs build/libYOUR_LIBRARY.a
ar: no archive members specified
usage:  ar -d [-TLsv] archive file ...
      ar -m [-TLsv] archive file ...
      ar -m [-abiTLsv] position archive file ...
      ar -p [-TLsv] archive [file ...]
      ar -q [-cTLsv] archive file ...
      ar -r [-cuTLsv] archive file ...
      ar -r [-abciuTLsv] position archive file ...
      ar -t [-TLsv] archive [file ...]
      ar -x [-ouTLsv] archive [file ...]
make: *** [build/libYOUR_LIBRARY.a] Error 1
$ make valgrind
VALGRIND="valgrind --log-file=/tmp/valgrind-%p.log" make
ar rcs build/libYOUR_LIBRARY.a
ar: no archive members specified
usage:  ar -d [-TLsv] archive file ...
      ar -m [-TLsv] archive file ...
      ar -m [-abiTLsv] position archive file ...
      ar -p [-TLsv] archive [file ...]
      ar -q [-cTLsv] archive file ...
      ar -r [-cuTLsv] archive file ...
      ar -r [-abciuTLsv] position archive file ...
      ar -t [-TLsv] archive [file ...]
      ar -x [-ouTLsv] archive [file ...]
make[1]: *** [build/libYOUR_LIBRARY.a] Error 1
make: *** [valgrind] Error 2
$
```

当我运行`clean:`目标时它会生效，但是由于我在`src/`目录中并没有任何源文件，其它命令并没有真正起作用。我会在下个练习中补完它。

附加题

- 尝试通过将源文件和头文件添加进`src/`，来使`Makefile`真正起作用，并且构建出库文件。在源文件中不应该需要`main`函数。
- 研究`check:`目标会使用`BADFUNCS`的正则表达式来寻找什么函数。
- 如果你没有做过自动测试，查询有关资料为以后做准备。

练习29：库和链接

> 原文：Exercise 29: Libraries And Linking
>
> 译者：飞龙

C语言编程的核心能力之一就是链接OS所提供的库。链接是一种为你的程序添加额外特性的方法，这些特性由其它人在系统中创建并打包。你已经使用了一些自动包含的标准库，但是我打算对库的不同类型和它们的作用做个解释。

首先，库在每个语言中都没有良好的设计。我不知道为什么，但是似乎语言的设计者都将链接视为不是特别重要的东西。它们通常令人混乱，难以使用，不能正确进行版本控制，并以不同的方式链接到各种地方。

C没有什么不同，但是C中的库和链接是Unix操作系统的组件，并且可执行的格式在很多年前就设计好了。学习C如何链接库有助于理解OS如何工作，以及它如何运行你的程序。

C中的库有两种基本类型：

静态

你可以使用 `ar` 和 `ranlib` 来构建它，就像上个练习中的 `libYOUR_LIBRARY.a` 那样（Windows下后缀为 `.lib` ）。这种库可以当做一系列 `.o` 对象文件和函数的容器，以及当你构建程序时，可以当做是一个大型的 `.o` 文件。

动态

它们通常以 `.so` （Linux）或 `.dll` （Windows）结尾。在OSX中，差不多有一百万种后缀，取决于版本和编写它的人。严格来讲，OSX中的 `.dylib` ， `.bundle` 和 `framework` 这三个之间没什么不同。这些文件都被构建好并且放置到指定的地方。当你运行程序时，OS会动态加载这些文件并且“凭空”链接到你的程序中。

我倾向于对小型或中型项目使用静态的库，因为它们易于使用，并且工作在在更多操作系统上。我也喜欢将所有代码放入静态库中，之后链接它来执行单元测试，或者链接到所需的程序中。

动态库适用于大型系统，它的空间十分有限，或者其中大量程序都使用相同的功能。这种情况下不应该为每个程序的共同特性静态链接所有代码，而是应该将它放到动态库中，这样它仅仅会为所有程序加载一份。

在上一个练习中，我讲解了如何构建静态库（ `.a` ），我会在本书的剩余部分用到它。这个练习中我打算向你展示如何构建一个简单的 `.so` 库，并且如何使用Unix系统的 `dlopen` 动态加载它。我会手动执行它，以便你可以理解每件实际发生的事情。之后，附加题这部分会使用c项目框架来创建它。

动态加载动态库

我创建了两个源文件来完成它。一个用于构建`libex29.so`库，另一个是个叫做`ex29`的程序，它可以加载这个库并运行其中的程序：

```
#include <stdio.h>
#include <ctype.h>
#include "dbg.h"


int print_a_message(const char *msg)
{
    printf("A STRING: %s\n", msg);

    return 0;
}


int uppercase(const char *msg)
{
    int i = 0;

    // BUG: \0 termination problems
    for(i = 0; msg[i] != '\0'; i++) {
        printf("%c", toupper(msg[i]));
    }

    printf("\n");

    return 0;
}

int lowercase(const char *msg)
{
    int i = 0;

    // BUG: \0 termination problems
    for(i = 0; msg[i] != '\0'; i++) {
        printf("%c", tolower(msg[i]));
    }

    printf("\n");

    return 0;
}

int fail_on_purpose(const char *msg)
{
    return 1;
}
```

这里面没什么神奇之处。其中故意留了一些bug，看你是否注意到了。你需要在随后修复它们。

我们将要使用`dlopen`，`dlsym`，和`dlclose`函数来处理上面的函数。

```
#include <stdio.h>
#include "dbg.h"
#include <dlfcn.h>

typedef int (*lib_function)(const char *data);


int main(int argc, char *argv[])
{
    int rc = 0;
    check(argc == 4, "USAGE: ex29 libex29.so function data");

    char *lib_file = argv[1];
    char *func_to_run = argv[2];
    char *data = argv[3];

    void *lib = dlopen(lib_file, RTLD_NOW);
    check(lib != NULL, "Failed to open the library %s: %s", lib_
file, dlerror());

    lib_function func = dlsym(lib, func_to_run);
    check(func != NULL, "Did not find %s function in the library
 %s: %s", func_to_run, lib_file, dlerror());

    rc = func(data);
    check(rc == 0, "Function %s return %d for data: %s", func_to
_run, rc, data);

    rc = dlclose(lib);
    check(rc == 0, "Failed to close %s", lib_file);

    return 0;

error:
    return 1;
}
```

我现在会拆分这个程序，便于你理解这一小段代码其中的原理。

ex29.c:5

我随后会使用这个函数指针定义，来调用库中的函数。这没什么新东西，确保你理解了它的作用。

ex29.c:17

在为一个小型程序做必要的初始化后，我使用了 `dlopen` 函数来加载由 `lib_file` 表示的库。这个函数返回一个句柄，我们随后会用到它，就像来打开文件那样。

ex29.c:18

如果出现错误，我执行了通常的检查并退出，但是要注意最后我使用了 `dlerror` 来查明发生了什么错误。

ex29.c:20

我使用了 `dlsym` 来获取 `lib` 中的函数，通过它的字面名称 `func_to_run` 。这是最强大的部分，因为我动态获取了一个函数指针，基于我从命令行 `argv` 获得的字符串。

ex29.c:23

接着我调用 `func` 函数，获得返回值并进行检查。

ex29.c:26

最后，我像关闭文件那样关闭了库。通常你需要在程序的整个运行期间保证它们打开，所以关闭操作并不非常实用，我只是在这里演示它。

> 译者注：由于能够使用系统调用加载，动态库可以被多种语言的程序调用，而静态库只能被C及兼容C的程序调用。

你会看到什么

既然你已经知道这些文件做什么了，下面是我的shell会话，用于构建 `libex29.so` 和 `ex29` 并随后运行它。下面的代码中你可以学到如何手动构建：

```
# compile the lib file and make the .so
# you may need -fPIC here on some platforms. add that if you get
 an error
$ cc -c libex29.c -o libex29.o
$ cc -shared -o libex29.so libex29.o

# make the loader program
$ cc -Wall -g -DNDEBUG ex29.c -ldl -o ex29

# try it out with some things that work
$ ex29 ./libex29.so print_a_message "hello there"
-bash: ex29: command not found
$ ./ex29 ./libex29.so print_a_message "hello there"
A STRING: hello there
$ ./ex29 ./libex29.so uppercase "hello there"
HELLO THERE
$ ./ex29 ./libex29.so lowercase "HELLO tHeRe"
hello there
$ ./ex29 ./libex29.so fail_on_purpose "i fail"
[ERROR] (ex29.c:23: errno: None) Function fail_on_purpose return
 1 for data: i fail

# try to give it bad args
$ ./ex29 ./libex29.so fail_on_purpose
[ERROR] (ex29.c:11: errno: None) USAGE: ex29 libex29.so function
 data

# try calling a function that is not there
$ ./ex29 ./libex29.so adfasfasdf asdfadff
[ERROR] (ex29.c:20: errno: None) Did not find adfasfasdf
  function in the library libex29.so: dlsym(0x1076009b0, adfasfa
sdf): symbol not found

# try loading a .so that is not there
$ ./ex29 ./libex.so adfasfasdf asdfadfas
[ERROR] (ex29.c:17: errno: No such file or directory) Failed to
open
    the library libex.so: dlopen(libex.so, 2): image not found
$
```

需要注意，你可能需要在不同OS、不同OS的不同版本，以及不同OS的不同版本的不同编译器上执行构建，则需要修改构建共享库的方式。如果我构建`libex29.so`的方式在你的平台上不起作用，请告诉我，我会为其它平台添加一些注解。

> 译者注：到处编写、到处调试、到处编译、到处发布。--vczh

> 注
>
> 有时候你像往常一样运行`cc -Wall -g -DNDEBUG -ldl ex29.c -o ex29`，并且认为它能够正常工作，但是没有。在一些平台上，参数的顺序会影响到它是否生效，这也没什么理由。在Debian或者Ubuntu中你需要执行`cc -Wall -g -DNDEBUG ex29.c -ldl -o ex29`，这是唯一的方式。所以虽然我在这里使用了OSX，但是以后如果你链接动态库的时候它找不到某个函数，要试着自己解决问题。
>
> 这里面比较麻烦的事情是，实际平台的不同会影响到命令参数的顺序。将`-ldl`放到某个位置没有理由与其它位置不同。它只是一个选项，还需要了解这些简直是太气人了。

如何使它崩溃

打开`lbex29.so`，并且使用能够处理二进制的编辑器编辑它。修改一些字节，然后关闭。看看你是否能使用`dlopen`函数来打开它，即使你修改了它。

附加题

- 你注意到我在`libex29.c`中写的不良代码了吗？我使用了一个`for`循环来检查`'\0'`的结尾，修改它们使这些函数总是接收字符串长度，并在函数内部使用。
- 使用项目框架目录，来为这个练习创建新的项目。将`libex29.c`放入`src/`目录，修改`Makefile`使它能够构建`build/libex29.so`。
- 将`ex29.c`改为`tests/ex29_tests.c`，使它做为单元测试执行。使它能够正常工作，意思是你需要修改它让它加载`build/libex29.so`文件，并且运行上面我手写的测试。
- 阅读`man dlopen`文档，并且查询所有有关函数。尝试`dlopen`的其它选项，比如`RTLD_NOW`。

练习30：自动化测试

> 原文：Exercise 30: Automated Testing
>
> 译者：飞龙

自动化测试经常用于例如Python和Ruby的其它语言，但是很少用于C。一部分原因是自动化加载和测试C的代码片段具有较高的难度。这一章中，我们会创建一个非常小型的测试“框架”，并且使用你的框架目录构建测试用例的示例。

我接下来打算使用，并且你会包含进框架目录的框架，叫做“minunit”，它以Jera Design所编写的一小段代码作为开始，之后我扩展了它，就像这样：

```
#undef NDEBUG
#ifndef _minunit_h
#define _minunit_h

#include <stdio.h>
#include <dbg.h>
#include <stdlib.h>

#define mu_suite_start() char *message = NULL

#define mu_assert(test, message) if (!(test)) { log_err(message)
; return message; }
#define mu_run_test(test) debug("\n-----%s", " " #test); \
    message = test(); tests_run++; if (message) return message;

#define RUN_TESTS(name) int main(int argc, char *argv[]) {\
    argc = 1; \
    debug("----- RUNNING: %s", argv[0]);\
        printf("----\nRUNNING: %s\n", argv[0]);\
        char *result = name();\
        if (result != 0) {\
            printf("FAILED: %s\n", result);\
        }\
        else {\
            printf("ALL TESTS PASSED\n");\
        }\
    printf("Tests run: %d\n", tests_run);\
        exit(result != 0);\
}


int tests_run;

#endif
```

原始的内容所剩不多了，现在我使用`dbg.h`宏，并且在模板测试运行器的末尾创建了大量的宏。在这小段代码中我们创建了整套函数单元测试系统，一旦它结合上shell脚本来运行测试，你可以将其用于你的C代码。

完成测试框架

为了基础这个练习，你应该让你的`src/libex29.c`正常工作，并且完成练习29的附加题，是`ex29.c`加载程序并合理运行。练习29中我这事了一个附加题来使它像单元测试一样工作，但是现在我打算重新想你展示如何使用`minunit.h`来做这件事。

首先我们需要创建一个简单的空单元测试，命名为`tests/libex29_tests.c`，在里面输入：

```
#include "minunit.h"

char *test_dlopen()
{

    return NULL;
}

char *test_functions()
{

    return NULL;
}

char *test_failures()
{

    return NULL;
}

char *test_dlclose()
{

    return NULL;
}

char *all_tests() {
    mu_suite_start();

    mu_run_test(test_dlopen);
    mu_run_test(test_functions);
    mu_run_test(test_failures);
    mu_run_test(test_dlclose);

    return NULL;
}

RUN_TESTS(all_tests);
```

这份代码展示了 `tests/minunit.h` 中的 `RUN_TESTS` 宏，以及如何使用其他的测试运行器宏。我没有编写实际的测试函数，所以你只能看到单元测试的结构。我首先会分解这个文件：

libex29_tests.c:1

包含 `minunit.h` 框架。

libex29_tests.c:3-7

第一个测试。测试函数具有固定的结构，它们不带任何参数并且返回 `char *` ，成功时为 `NULL` 。这非常重要，因为其他宏用于向测试运行器返回错误信息。

libex29_tests.c:9-25

与第一个测试相似的更多测试。

libex29_tests.c:27

控制其他测试的运行器函数。它和其它测试用例格式一致，但是使用额外的东西来配置。

libex29_tests.c:28

为 `mu_suite_start` 测试设置一些通用的东西。

libex29_tests.c:30

这就是使用 `mu_run_test` 返回结果的地方。

libex29_tests.c:35

在你运行所有测试之后，你应该返回 `NULL` ，就像普通的测试函数一样。

libex29_tests.c:38

最后需要使用 `RUN_TESTS` 宏来启动 `main` 函数，让它运行 `all_tests` 启动器。

这就是用于运行测试所有代码了，现在你需要尝试使它运行在项目框架中。下面是我的执行结果：

```
not printable
```

我首先执行 `make clean` ，之后我运行了构建，它将模板改造为 `libYOUR_LIBRARY.a` 和 `libYOUR_LIBRARY.so` 文件。要记住你需要在练习29的附加题中完成它。但如果你没有完成的话，下面是我所使用的 `Makefile` 的文件差异：

```
diff --git a/code/c-skeleton/Makefile b/code/c-skeleton/Makefile
index 135d538..21b92bf 100644
--- a/code/c-skeleton/Makefile
+++ b/code/c-skeleton/Makefile
@@ -9,9 +9,10 @@ TEST_SRC=$(wildcard tests/*_tests.c)
 TESTS=$(patsubst %.c,%,$(TEST_SRC))

 TARGET=build/libYOUR_LIBRARY.a
+SO_TARGET=$(patsubst %.a,%.so,$(TARGET))

 # The Target Build
-all: $(TARGET) tests
+all: $(TARGET) $(SO_TARGET) tests

 dev: CFLAGS=-g -Wall -Isrc -Wall -Wextra $(OPTFLAGS)
 dev: all
@@ -21,6 +22,9 @@ $(TARGET): build $(OBJECTS)
        ar rcs $@ $(OBJECTS)
        ranlib $@

+$(SO_TARGET): $(TARGET) $(OBJECTS)
+       $(CC) -shared -o $@ $(OBJECTS)
+
 build:
        @mkdir -p build
        @mkdir -p bin
```

完成这些改变后，你现在应该能够构建任何东西，并且你可以最后补完剩余的单元测试函数：

```
#include "minunit.h"
#include <dlfcn.h>

typedef int (*lib_function)(const char *data);
char *lib_file = "build/libYOUR_LIBRARY.so";
void *lib = NULL;

int check_function(const char *func_to_run, const char *data, int
 expected)
{
    lib_function func = dlsym(lib, func_to_run);
    check(func != NULL, "Did not find %s function in the library
 %s: %s", func_to_run, lib_file, dlerror());

    int rc = func(data);
    check(rc == expected, "Function %s return %d for data: %s",
func_to_run, rc, data);

    return 1;
error:
```

```
    return 0;
}

char *test_dlopen()
{
    lib = dlopen(lib_file, RTLD_NOW);
    mu_assert(lib != NULL, "Failed to open the library to test."
);

    return NULL;
}

char *test_functions()
{
    mu_assert(check_function("print_a_message", "Hello", 0), "pr
int_a_message failed.");
    mu_assert(check_function("uppercase", "Hello", 0), "uppercas
e failed.");
    mu_assert(check_function("lowercase", "Hello", 0), "lowercas
e failed.");

    return NULL;
}

char *test_failures()
{
    mu_assert(check_function("fail_on_purpose", "Hello", 1), "fa
il_on_purpose should fail.");

    return NULL;
}

char *test_dlclose()
{
    int rc = dlclose(lib);
    mu_assert(rc == 0, "Failed to close lib.");

    return NULL;
}

char *all_tests() {
    mu_suite_start();

    mu_run_test(test_dlopen);
    mu_run_test(test_functions);
    mu_run_test(test_failures);
    mu_run_test(test_dlclose);

    return NULL;
}

RUN_TESTS(all_tests);
```

我希望你可以弄清楚它都干了什么，因为这里没有什么新的东西，除了 `check_function` 函数。这是一个通用的模式，其中我需要重复执行一段代码，然后通过为之创建宏或函数来使它自动化。这里我打算运行 `.so` 中所加载的函数，所以我创建了一个小型函数来完成它。

附加题

- 这段代码能起作用，但是可能有点乱。清理框架目录，是它包含所有这些文件，但是移除任何和练习29有关的代码。你应该能够复制这个目录并且无需很多编辑操作就能开始新的项目。
- 研究 `runtests.sh` ，并且查询有关 `bash` 语法的资料，来弄懂它的作用。你能够编写这个脚本的C版本吗？

练习31：代码调试

> 原文：[Exercise 31: Debugging Code]
>
> 译者：飞龙

我已经教给你一些关于我的强大的调试宏的技巧，并且你已经开始用它们了。当我调试代码时，我使用 `debug()` 宏，分析发生了什么以及跟踪问题。在这个练习中我打算教给你一些使用gdb的技巧，用于监视一个不会退出的简单程序。你会学到如何使用gdb附加到运行中的进程，并挂起它来观察发生了什么。在此之后我会给你一些用于gdb的小提示和小技巧。

调试输出、GDB或Valgrind

我主要按照一种“科学方法”的方式来调试，我会提出可能的所有原因，之后排除它们或证明它们导致了缺陷。许多程序员拥有的问题是它们对解决bug的恐慌和急躁使他们觉得这种方法会“拖慢”他们。它们并没有注意到，它们已经失败了，并且在收集无用的信息。我发现日志（调试输出）会强迫我科学地解决bug，并且在更多情况下易于收集信息。

此外，使用调试输出来作为我的首要调试工具的理由如下：

- 你可以使用变量的调试输出，来看到程序执行的整个轨迹，它让你跟踪变量是如何产生错误的。使用gdb的话，你必须为每个变量放置查看和调试语句，并且难以获得执行的实际轨迹。
- 调试输出存在于代码中，当你需要它们是你可以重新编译使它们回来。使用gdb的话，你每次调试都需要重新配置相同的信息。
- 当服务器工作不正常时，它的调试日志功能易于打开，并且在它运行中可以监视日志来查看哪里不对。系统管理员知道如何处理日志，他们不知道如何使用gdb。
- 打印信息更加容易。调试器通常由于它奇特的UI和前后矛盾显得难用且古怪。 `debug("Yo, dis right? %d", my_stuff);` 就没有那么麻烦。
- 编写调试输出来发现缺陷，强迫你实际分析代码，并且使用科学方法。你可以认为它是，“我假设这里的代码是错误的”，你可以运行它来验证你的假设，如果这里没有错误那么你可以移动到其它地方。这看起来需要更长时间，但是实际上更快，因为你经历了“鉴别诊断”的过程，并排除所有可能的原因，直到你找到它。
- 调试输入更适于和单元测试一起运行。你可以实际上总是编译调试语句，单元测试时可以随时查看日志。如果你用gdb，你需要在gdb中重复运行单元测试，并跟踪他来查看发生了什么。
- 使用Valgrind可以得到和调试输出等价的内存相关的错误，所以你并不需要使用类似gdb的东西来寻找缺陷。

尽管所有原因显示我更倾向于 `debug` 而不是 `gdb` ，我还是在少数情况下回用到 `gdb` ，并且我认为你应该选择有助于你完成工作的工具。有时，你只能够连接到一个崩溃的程序并且四处转悠。或者，你得到了一个会崩溃的服务器，你只能够

获得一些核心文件来一探究竟。这些货少数其它情况中，gdb是很好的办法。你最好准备尽可能多的工具来解决问题。

接下来我会通过对比gdb、调试输出和Valgrind来详细分析，像这样：

- Valgrind用于捕获所有内存错误。如果Valgrind中含有错误或Valgrind会严重拖慢程序，我会使用gdb。
- 调试输出用于诊断或修复有关逻辑或使用上的缺陷。在你使用Valgrind之前，这些共计90%的缺陷。
- 使用gdb解决剩下的“谜之bug”，或如要收集信息的紧急情况。如果Valgrind不起作用，并且我不能打印出所需信息，我就会使用gdb开始四处搜索。这里我仅仅使用gdb来收集信息。一旦我弄清发生了什么，我会回来编程单元测试来引发缺陷，之后编程打印语句来查找原因。

调试策略

这一过程适用于你打算使用任何调试技巧，无论是Valgrind、调试输出，或者使用调试器。我打算以使用 `gdb` 的形式来描述他，因为似乎人们在使用调试器是会跳过它。但是应当对每个bug使用它，直到你只需要在非常困难的bug上用到。

- 创建一个小型文本文件叫做 `notes.txt` ，并且将它用作记录想法、bug和问题的“实验记录”。
- 在你使用 `gdb` 之前，写下你打算修复的bug，以及可能的产生原因。
- 对于每个原因，写下你所认为的，问题来源的函数或文件，或者仅仅写下你不知道。
- 现在启动 `gdb` 并且使用 `file:function` 挑选最可能的因素，之后在那里设置断点。
- 使用 `gdb` 运行程序，并且确认它是否是真正原因。查明它的最好方式就是看看你是否可以使用 `set` 命令，简单修复问题或者重现错误。
- 如果它不是真正原因，则在 `notes.txt` 中标记它不是，以及理由。移到下一个可能的原因，并且使最易于调试的，之后记录你收集到的信息。

这里你并没有注意到，它是最基本的科学方法。你写下一些假设，之后调试来证明或证伪它们。这让你洞察到更多可能的因素，最终使你找到他。这个过程有助于你避免重复步入同一个可能的因素，即使你发现它们并不可能。

你也可以使用调试输出来执行这个过程。唯一的不同就是你实际在源码中编写假设来推测问题所在，而不是 `notes.txt` 中。某种程度上，调试输出强制你科学地解决bug，因为你需要将假写为打印语句。

使用 GDB

我将在这个练习中调试下面这个程序，它只有一个不会正常终止的 `while` 循环。我在里面放置了一个 `usleep` 调用，使它循环起来更加有趣。

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
    int i = 0;

    while(i < 100) {
        usleep(3000);
    }

    return 0;
}
```

像往常一样编译，并且在`gdb`下启动它，例如：`gdb ./ex31`。

一旦它运行之后，我打算让你使用这些`gdb`命令和它交互，并且观察它们的作用以及如何使用它们。

help COMMAND

获得`COMMAND`的简单帮助。

break file.c:(line|function)

在你希望暂停之星的地方设置断点。你可以提供行号或者函数名称，来在文件中的那个地方暂停。

run ARGS

运行程序，使用`ARGS`作为命令行参数。

cont

继续执行程序，直到断点或错误。

step

单步执行代码，但是会进入函数内部。使用它来跟踪函数内部，来观察它做了什么。

next

就像是`step`，但是他会运行函数并步过它们。

backtrace (or bt)

执行“跟踪回溯”，它会转储函数到当前执行点的执行轨迹。对于查明如何执行到这里非常有用，因为它也打印出传给每个函数的参数。它和Valgrind报告内存错误的方式很接近。

set var X = Y

将变量 `X` 设置为 `Y` 。

print X

打印出 `X` 的值，你通常可以使用C的语法来访问指针的值或者结构体的内容。

ENTER

重复上一条命令。

quit

退出 `gdb` 。

这些都是我使用 `gdb` 时的主要命令。你现在的任务是玩转它们和 `ex31` ，你会对它的输出更加熟悉。

一旦你熟悉了 `gdb` 之后，你会希望多加使用它。尝试在更复杂的程序，例如 `devpkg` 上使用它，来观察你是否能够改函数的执行或分析出程序在做什么。

附加到进程

`gdb` 最实用的功能就是附加到运行中的程序，并且就地调试它的能力。当你拥有一个崩溃的服务器或GUI程序，你通常不需要像之前那样在 `gdb` 下运行它。而是可以直接启动它，希望它不要马上崩溃，之后附加到它并设置断点。练习的这一部分中我会向你展示怎么做。

当你退出 `gdb` 之后，如果你停止了 `ex31` 我希望你重启它，之后开启另一个中断窗口以便于启动 `gdb` 并附加。进程附加就是你让 `gdb` 连接到已经运行的程序，以便于你实时监测它。它会挂起程序来让你单步执行，当你执行完之后程序会像往常一样恢复运行。

下面是一段会话，我对 `ex31` 做了上述事情，单步执行它，之后修改 `while` 循环并使它退出。

```
$ ps ax | grep ex31
10026 s000  S+     0:00.11 ./ex31
10036 s001  R+     0:00.00 grep ex31

$ gdb ./ex31 10026
GNU gdb 6.3.50-20050815 (Apple version gdb-1705) (Fri Jul  1 10:
50:06 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
 and you are
welcome to change it and/or distribute copies of it under certai
n conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" f
or details.
```

```
This GDB was configured as "x86_64-apple-darwin"...Reading symbo
ls for shared libraries .. done

/Users/zedshaw/projects/books/learn-c-the-hard-way/code/10026: N
o such file or directory
Attaching to program: `/Users/zedshaw/projects/books/learn-c-the
-hard-way/code/ex31', process 10026.
Reading symbols for shared libraries + done
Reading symbols for shared libraries ++........................
done
Reading symbols for shared libraries + done
0x00007fff862c9e42 in __semwait_signal ()

(gdb) break 8
Breakpoint 1 at 0x107babf14: file ex31.c, line 8.

(gdb) break ex31.c:11
Breakpoint 2 at 0x107babf1c: file ex31.c, line 12.

(gdb) cont
Continuing.

Breakpoint 1, main (argc=1, argv=0x7fff677aabd8) at ex31.c:8
8       while(i < 100) {

(gdb) p i
$1 = 0

(gdb) cont
Continuing.

Breakpoint 1, main (argc=1, argv=0x7fff677aabd8) at ex31.c:8
8       while(i < 100) {

(gdb) p i
$2 = 0

(gdb) list
3
4   int main(int argc, char *argv[])
5   {
6       int i = 0;
7
8       while(i < 100) {
9           usleep(3000);
10      }
11
12      return 0;

(gdb) set var i = 200

(gdb) p i
$3 = 200
```

```
(gdb) next

Breakpoint 2, main (argc=1, argv=0x7fff677aabd8) at ex31.c:12
12      return 0;

(gdb) cont
Continuing.

Program exited normally.
(gdb) quit
$
```

> 注
>
> 在OSX上你可能会看到输入root密码的GUI输入框，并且即使你输入了密码还是会得到来自 `gdb` 的“Unable to access task for process-id XXX: (os/kern) failure.”的错误。这种情况下，你需要停止 `gdb` 和 `ex31` 程序，并重新启动程序使它工作，只要你成功输入了root密码。

我会遍历整个会话，并且解释我做了什么：

gdb:1

使用 `ps` 来寻找我想要附加的 `ex31` 的进程ID。

gdb:5

我使用 `gdb ./ex31 PID` 来附加到进程，其中 `PID` 替换为我所拥有的进程ID。

gdb:6-19

`gdb` 打印出了一堆关于协议的信息，接着它读取了所有东西。

gdb:21

程序被附加，并且在当前执行点上停止。所以现在我在文件中的第8行使用 `break` 设置了断点。我假设我这么做的时候，已经在这个我想中断的文件中了。

gdb:24

执行 `break` 的更好方式，是提供 `file.c line` 的格式，便于你确保定位到了正确的地方。我在这个 `break` 中这样做。

gdb:27

我使用 `cont` 来继续运行，直到我命中了断点。

gdb:30-31

我已到达断点，于是 `gdb` 打印出我需要了解的变量（ `argc` 和 `argv` ），以及停下来的位置，之后打印出断点的行号。

gdb:33-34

我使用 `print` 的缩写 `p` 来打印出 `i` 变量的值，它是0。

gdb:36

继续运行来查看 `i` 是否改变。

gdb:42

再次打印出 `i` ，显然它没有变化。

gdb:45-55

使用 `list` 来查看代码是什么，之后我意识到它不可能退出，因为我没有自增 `i` 。

gdb:57

确认我的假设是正确的，即 `i` 需要使用 `set` 命令来修改为 `i = 200` 。这是 `gdb` 最优秀的特性之一，让你“修改”程序来让你快速知道你是否正确。

gdb:59

打印 `i` 来确保它已改变。

gdb:62

使用 `next` 来移到下一段代码，并且我发现命中了 `ex31.c:12` 的断点，所以这意味着 `while` 循环已退出。我的假设正确，我需要修改 `i` 。

gdb:67

使用 `cont` 来继续运行，程序像往常一样退出。

gdb:71

最后我使用 `quit` 来退出 `gdb` 。

GDB 技巧

下面是你可以用于GDB的一些小技巧：

gdb --args

通常 `gdb` 获得你提供的变量并假设它们用于它自己。使用 `--args` 来向程序传递它们。

thread apply all bt

转储所有线程的执行轨迹，非常有用。

gdb --batch --ex r --ex bt --ex q --args

运行程序，当它崩溃时你会得到执行轨迹。

?

如果你有其它技巧，在评论中写下它吧。

附加题

- 找到一个图形化的调试器，将它与原始的 `gdb` 相比。它们在本地调试程序时非常有用，但是对于在服务器上调试没有任何意义。
- 你可以开启OS上的“核心转储”，当程序崩溃时你会得到一个核心文件。这个核心文件就像是对程序的解剖，便于你了解崩溃时发生了什么，以及由什么原因导致。修改 `ex31.c` 使它在几个迭代之后崩溃，之后尝试得到它的核心转储并分析。

练习32：双向链表

> 原文：Exercise 32: Double Linked Lists
>
> 译者：飞龙

这本书的目的是教给你计算机实际上如何工作，这也包括多种数据结构和算法函数。计算机自己其实并没有太大用处。为了让它们做一些有用的事情，你需要构建数据，之后在这些结构上组织处理。其它编程语言带有实现所有这些结构的库，或者带有直接的语法来创建它们。C需要你手动实现所有数据结构，这使它成为最“完美”的语言，让你知道它们的工作原理。

我的目标是交给你这些数据结构，以及相关算法的知识，来帮助你完成下面这三件事：

- 理解Python、Ruby或JavaScript的 `data = {"name": "Zed"}` 到底做了什么。
- 使用数据结构来解决问题，使你成为更好的C程序员。
- 学习数据结构和算法的核心部分，让你知道在特定条件下哪个最好。

数据结构是什么。

“数据结构”这个名称自己就能够解释。它是具有特性模型的数据组织方法。这一模型可能设计用于以新的方法处理数据，也可能只是用于将它们更高效地储存在磁盘上。这本书中我会遵循一些简单的模式来构建可用的数据结构：

- 定义一个结构的主要“外部结构”。
- 定义一个结构的内容，通常是带有链接的节点。
- 创建函数操作它们的函数。

C中还有其它样式的数据结构，但是这个模式效果很好，并且对于你创建的大部分数据结构都适用。

构建库

对于这本书的剩余部分，当你完成这本书之后，你将会创建一个可用的库。这个库会包含下列元素：

- 为每个数据结构编写的头文件 `.h` 。
- 为算法编写的实现文件 `.c` 。
- 用于测试它们确保有效的单元测试。
- 从头文件自动生成的文档。

你已经实现了 `c-skeleton` （项目框架目录），使用它来创建一个 `liblcthw` 项目：

```
$ cp -r c-skeleton liblcthw
$ cd liblcthw/
$ ls
LICENSE                Makefile        README.md       bin
    build              src             tests
$ vim Makefile
$ ls src/
dbg.h                  libex29.c       libex29.o
$ mkdir src/lcthw
$ mv src/dbg.h src/lcthw
$ vim tests/minunit.h
$ rm src/libex29.* tests/libex29*
$ make clean
rm -rf build  tests/libex29_tests
rm -f tests/tests.log
find . -name "*.gc*" -exec rm {} \;
rm -rf `find . -name "*.dSYM" -print`
$ ls tests/
minunit.h  runtests.sh
$
```

这个会话中我执行了下列事情：

- 复制了 `c-skeleton` 。
- 编辑Makefile，将 `libYOUR_LIBRARY.a` 改为 `liblcthw.a` 作为新的 `TARGET` 。
- 创建 `src/lcthw` 目录，我们会在里面放入代码。
- 移动 `src/dbg.h` 文件到新的目录中。
- 编辑 `tests/minunit.h` ，使它使用所包含的 `#include <lcthw/dbg.h>` 。
- 移除 `libex29.*` 中我们不需要的源文件和测试文件。
- 清理所有遗留的东西。

执行完之后你就准备好开始构建库了，我打算构建第一个数据结构是双向链表。

双向链表

我们将要向 `liblcthw` 添加的第一个数据结构是双向链表。这是你能够构建的最简单的数据结构，并且它拥有针对特定操作的实用属性。单向链表通过指向下一个或上一个元素的节点来工作。“双向”链表持有全部这两个指针，而“单向”链表只持有下一个元素的指针。

由于每个节点都有下一个和上一个元素的指针，并且你可以跟踪联保的第一个和最后的元素，你就可以快速地执行一些操作。任何涉及到插入和删除元素的操作会非常快。它对大多数人来说也易于实现。

链表的主要缺点是，遍历它涉及到处理沿途每个单个的指针。这意味着搜索、多数排序以及迭代元素会表较慢。这也意味着你不能直接跳过链表的随机一部分。如果换成数组，你就可以直接索引到它的中央，但是链表不行。也就是说如果你想要访

问第十个元素，你必须经过1~9。

定义

正如在这个练习的介绍部分所说，整个过程的第一步，是编程一个头文件，带有正确的C结构定义。

```
#ifndef lcthw_List_h
#define lcthw_List_h

#include <stdlib.h>

struct ListNode;

typedef struct ListNode {
    struct ListNode *next;
    struct ListNode *prev;
    void *value;
} ListNode;

typedef struct List {
    int count;
    ListNode *first;
    ListNode *last;
} List;

List *List_create();
void List_destroy(List *list);
void List_clear(List *list);
void List_clear_destroy(List *list);

#define List_count(A) ((A)->count)
#define List_first(A) ((A)->first != NULL ? (A)->first->value :
NULL)
#define List_last(A) ((A)->last != NULL ? (A)->last->value : NUL
L)

void List_push(List *list, void *value);
void *List_pop(List *list);

void List_unshift(List *list, void *value);
void *List_shift(List *list);

void *List_remove(List *list, ListNode *node);

#define LIST_FOREACH(L, S, M, V) ListNode *_node = NULL;\
    ListNode *V = NULL;\
    for(V = _node = L->S; _node != NULL; V = _node = _node->M)

#endif
```

我所做的第一件事就是创建两个结构，`ListNode`和包含这些节点的`List`。这创建了是将在函数中使用的数据结构，以及随后定义的宏。如果你浏览这些函数，它们看起来非常简单。当我讲到实现时，我会解释他们，但我更希望你能猜出它们的作用。

这些数据结构的工作方式，就是每个`ListNode`都有三个成员。

- 值，它是无类型的指针，存储我们想在链表中放置的东西。
- `ListNode *next`指针，它指向另一个储存下一个元素的`ListNode`。
- `ListNode *prev`指针，它指向另一个储存上一个元素的`ListNode`。

`List`结构只是这些`ListNode`结构的容器，它们互联链接组成链型。它跟踪链表的`count`，`first`和`last`元素。

最后，看一看`src/lcthw/list.h:37`，其中我定义了`LIST_FOREACH`宏。这是个常见的习语，你可以创建一个宏来生成迭代代码，使用者就不会弄乱了。正确使用这类执行过程来处理数据结构十分困难，所以可以编写宏来帮助使用者。当我讲到实现时，你可以看到我如何使用它。

实现

一旦你理解了它们之后，你很可能理解了双向链表如何工作。它只是带有两个指针的节点，指向链表中前一个和后一个元素。接下来你可以编写`src/lcthw/list.c`中的代码，来理解每个操作如何实现。

```c
#include <lcthw/list.h>
#include <lcthw/dbg.h>

List *List_create()
{
    return calloc(1, sizeof(List));
}

void List_destroy(List *list)
{
    LIST_FOREACH(list, first, next, cur) {
        if(cur->prev) {
            free(cur->prev);
        }
    }

    free(list->last);
    free(list);
}


void List_clear(List *list)
{
    LIST_FOREACH(list, first, next, cur) {
        free(cur->value);
```

```
    }
}


void List_clear_destroy(List *list)
{
    List_clear(list);
    List_destroy(list);
}


void List_push(List *list, void *value)
{
    ListNode *node = calloc(1, sizeof(ListNode));
    check_mem(node);

    node->value = value;

    if(list->last == NULL) {
        list->first = node;
        list->last = node;
    } else {
        list->last->next = node;
        node->prev = list->last;
        list->last = node;
    }

    list->count++;

error:
    return;
}

void *List_pop(List *list)
{
    ListNode *node = list->last;
    return node != NULL ? List_remove(list, node) : NULL;
}

void List_unshift(List *list, void *value)
{
    ListNode *node = calloc(1, sizeof(ListNode));
    check_mem(node);

    node->value = value;

    if(list->first == NULL) {
        list->first = node;
        list->last = node;
    } else {
        node->next = list->first;
        list->first->prev = node;
        list->first = node;
```

```
    }

    list->count++;

error:
    return;
}

void *List_shift(List *list)
{
    ListNode *node = list->first;
    return node != NULL ? List_remove(list, node) : NULL;
}

void *List_remove(List *list, ListNode *node)
{
    void *result = NULL;

    check(list->first && list->last, "List is empty.");
    check(node, "node can't be NULL");

    if(node == list->first && node == list->last) {
        list->first = NULL;
        list->last = NULL;
    } else if(node == list->first) {
        list->first = node->next;
        check(list->first != NULL, "Invalid list, somehow got a
first that is NULL.");
        list->first->prev = NULL;
    } else if (node == list->last) {
        list->last = node->prev;
        check(list->last != NULL, "Invalid list, somehow got a n
ext that is NULL.");
        list->last->next = NULL;
    } else {
        ListNode *after = node->next;
        ListNode *before = node->prev;
        after->prev = before;
        before->next = after;
    }

    list->count--;
    result = node->value;
    free(node);

error:
    return result;
}
```

我实现了双向链表上的所有操作，它们不能用简单的宏来完成。比起覆盖文件中的每一行，我打算为 `list.h` 和 `list.c` 中的每个操作提供一个高阶的概览。你需要自己阅读代码。

list.h:List_count

返回链表中元素数量，它在元素添加或移除时维护。

list.h:List_first

返回链表的首个元素，但是并不移除它。

list.h:List_last

返回链表的最后一个元素，但是不移除它。

list.h:LIST_FOREACH

遍历链表中的元素。

list.c:List_create

简单地创建主要的`List`结构。

list.c:List_destroy

销毁`List`以及其中含有的所有元素。

list.c:List_clear

为释放每个节点中的值（而不是节点本身）创建的辅助函数。

list.c:List_clear_destroy

清理并销毁链表。它并不十分搞笑因为它对每个元素遍历两次。

list.c:List_push

第一个操作演示了链表的有点。它向链表尾添加新的元素，由于只是一些指针赋值，所以非常快。

list.c:List_pop

`List_push`的反向版本，它去除最后一个元素并返回它。

list.c:List_unshift

亦可以轻易对链表执行的另一件事，就是快速地向链表头部添加元素。由于找不到合适的词，这里我把它称为`unshift`。

list.c:List_shift

类似`List_pop`，但是它移除链表的首个元素并返回。

list.c:List_remove

当你执行`List_pop`或`List_shift`时，它执行实际的移除操作。在数据结构中移除数据总是看似比较困难，这个函数也不例外。它需要处理一些条件，取决于被移除的位置，在开头、在结尾、开头并且结尾，或者在中间。

这些函数大多数都没什么特别的，你应该能够轻易描述出来，并且根据代码来理解它。你应该完全专注于 `List_destroy` 中的 `LIST_FOREACH` 如何使用来理解它如何简化通常的操作。

测试

在你编译它们之前，需要创建测试来确保它们正确执行。

```c
#include "minunit.h"
#include <lcthw/list.h>
#include <assert.h>

static List *list = NULL;
char *test1 = "test1 data";
char *test2 = "test2 data";
char *test3 = "test3 data";


char *test_create()
{
    list = List_create();
    mu_assert(list != NULL, "Failed to create list.");

    return NULL;
}


char *test_destroy()
{
    List_clear_destroy(list);

    return NULL;

}


char *test_push_pop()
{
    List_push(list, test1);
    mu_assert(List_last(list) == test1, "Wrong last value.");

    List_push(list, test2);
    mu_assert(List_last(list) == test2, "Wrong last value");

    List_push(list, test3);
    mu_assert(List_last(list) == test3, "Wrong last value.");
    mu_assert(List_count(list) == 3, "Wrong count on push.");

    char *val = List_pop(list);
    mu_assert(val == test3, "Wrong value on pop.");
```

```
    val = List_pop(list);
    mu_assert(val == test2, "Wrong value on pop.");

    val = List_pop(list);
    mu_assert(val == test1, "Wrong value on pop.");
    mu_assert(List_count(list) == 0, "Wrong count after pop.");

    return NULL;
}

char *test_unshift()
{
    List_unshift(list, test1);
    mu_assert(List_first(list) == test1, "Wrong first value.");

    List_unshift(list, test2);
    mu_assert(List_first(list) == test2, "Wrong first value");

    List_unshift(list, test3);
    mu_assert(List_first(list) == test3, "Wrong last value.");
    mu_assert(List_count(list) == 3, "Wrong count on unshift.");

    return NULL;
}

char *test_remove()
{
    // we only need to test the middle remove case since push/sh
ift
    // already tests the other cases

    char *val = List_remove(list, list->first->next);
    mu_assert(val == test2, "Wrong removed element.");
    mu_assert(List_count(list) == 2, "Wrong count after remove."
);
    mu_assert(List_first(list) == test3, "Wrong first after remo
ve.");
    mu_assert(List_last(list) == test1, "Wrong last after remove
.");

    return NULL;
}


char *test_shift()
{
    mu_assert(List_count(list) != 0, "Wrong count before shift."
);

    char *val = List_shift(list);
    mu_assert(val == test3, "Wrong value on shift.");
```

```
    val = List_shift(list);
    mu_assert(val == test1, "Wrong value on shift.");
    mu_assert(List_count(list) == 0, "Wrong count after shift.")
;

    return NULL;
}



char *all_tests() {
    mu_suite_start();

    mu_run_test(test_create);
    mu_run_test(test_push_pop);
    mu_run_test(test_unshift);
    mu_run_test(test_remove);
    mu_run_test(test_shift);
    mu_run_test(test_destroy);

    return NULL;
}

RUN_TESTS(all_tests);
```

它简单地遍历了每个操作，并且确保它们有效。我在测试中做了简化，对于整个程序我只创建了一个 `List *list` ，这解决了为每个测试构建一个 `List` 的麻烦，但它同时意味着一些测试会受到之前测试的影响。这里我试着是每个测试不改变链表，或实际使用上一个测试的结果。

你会看到什么

如果你正确完成了每件事，当你执行构建并且运行单元测试是，你会看到：

```
$ make
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  -fPIC   -c -o
src/lcthw/list.o src/lcthw/list.c
ar rcs build/liblcthw.a src/lcthw/list.o
ranlib build/liblcthw.a
cc -shared -o build/liblcthw.so src/lcthw/list.o
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  build/liblcthw
.a    tests/list_tests.c   -o tests/list_tests
sh ./tests/runtests.sh
Running unit tests:
----
RUNNING: ./tests/list_tests
ALL TESTS PASSED
Tests run: 6
tests/list_tests PASS
$
```

确保6个测试运行完毕，以及构建时没有警告或错误，并且成功构建了 `build/liblcthw.a` 和 `build/liblcthw.so` 文件。

如何改进

我打算告诉你如何改进代码，而不是使它崩溃。

- 你可以使用 `LIST_FOREACH` 并在循环中调用 `free` 来使 `List_clear_destroy` 更高效。
- 你可以为一些先决条件添加断言，使其部结构 `NULL` 值作为 `List *list` 的参数。
- 你可以添加不变了，来检查列表的内容始终正确，例如 `count` 永远不会 `< 0` ，如果 `count > 0` ， `first` 不为 `NULL` 。
- 你可以向头文件添加文档，在每个结构、函数和宏之前添加描述其作用的注释。

这些改进执行了防御性编程实践，并且“加固”了代码来避免错误或使用不当。马上去做这些事情，之后找到尽可能多的办法来改进代码。

附加题

- 研究双向和单向链表，以及什么情况下其中一种优于另一种。
- 研究双向链表的限制。例如，虽然它们对于插入和删除元素很高效，但是对于变量元素比较慢。
- 还缺少什么你能想到的操作？比如复制、连接、分割等等。实现这些操作，并且为它们编写单元测试。

练习33：链表算法

> 原文：Exercise 33: Linked List Algorithms
>
> 译者：飞龙

我将想你介绍涉及到排序的两个算法，你可以用它们操作链表。我首先要警告你，如果你打算对数据排序，不要使用链表，它们对于排序十分麻烦，并且有更好的数据结构作为替代。我向你介绍这两种算法只是因为它们难以在链表上完成，并且让你思考如何高效操作它们。

为了编写这本书，我打算将算法放在两个不同的文件中，`list_algos.h`和`list_algos.c`，之后在`list_algos_test.c`中编写测试。现在你要按照我的结构，因为它足以把事情做好，但是如果你使用其它的库要记住这并不是通用的结构。

这个练习中我打算给你一些额外的挑战，并且希望你不要作弊。我打算先给你单元测试，并且让你打下来。之后让你基于它们在维基百科中的描述，尝试实现这个两个算法，之后看看你的代码是否和我的类似。

冒泡排序和归并排序

互联网的强大之处，就是我可以仅仅给你冒泡排序和归并排序的链接，来让你学习它们。是的，这省了我很多字。现在我要告诉你如何使用它们的伪代码来实现它们。你可以像这样来实现算法：

- 阅读描述，并且观察任何可视化的图表。
- 使用方框和线条在纸上画出算法，或者使用一些带有数字的卡片（比如扑克牌），尝试手动执行算法。这会向你形象地展示算法的执行过程。
- 在`list_algos.c`文案总创建函数的主干，并且创建`list_algos.h`文件，之后创建测试代码。
- 编写第一个测试并且编译所有东西。
- 回到维基百科页面，复制粘贴伪代码到你创建的函数中（不是C代码）。
- 将伪代码翻译成良好的C代码，就像我教你的那样，使用你的单元测试来保证它有效。
- 为边界情况补充一些测试，例如空链表，排序号的链表，以及其它。
- 对下一个算法重复这些过程并测试。

我只是告诉你理解大多数算法的秘密，直到你碰到一些更加麻烦的算法。这里你只是按照维基百科来实现冒泡排序和归并排序，它们是一个好的起始。

单元测试

下面是你应该通过的单元测试：

```
#include "minunit.h"
#include <lcthw/list_algos.h>
#include <assert.h>
#include <string.h>

char *values[] = {"XXXX", "1234", "abcd", "xjvef", "NDSS"};
#define NUM_VALUES 5

List *create_words()
{
    int i = 0;
    List *words = List_create();

    for(i = 0; i < NUM_VALUES; i++) {
        List_push(words, values[i]);
    }

    return words;
}

int is_sorted(List *words)
{
    LIST_FOREACH(words, first, next, cur) {
        if(cur->next && strcmp(cur->value, cur->next->value) > 0
) {
            debug("%s %s", (char *)cur->value, (char *)cur->next
->value);
            return 0;
        }
    }

    return 1;
}

char *test_bubble_sort()
{
    List *words = create_words();

    // should work on a list that needs sorting
    int rc = List_bubble_sort(words, (List_compare)strcmp);
    mu_assert(rc == 0, "Bubble sort failed.");
    mu_assert(is_sorted(words), "Words are not sorted after bubb
le sort.");

    // should work on an already sorted list
    rc = List_bubble_sort(words, (List_compare)strcmp);
    mu_assert(rc == 0, "Bubble sort of already sorted failed.");
    mu_assert(is_sorted(words), "Words should be sort if already
 bubble sorted.");

    List_destroy(words);
```

```
    // should work on an empty list
    words = List_create(words);
    rc = List_bubble_sort(words, (List_compare)strcmp);
    mu_assert(rc == 0, "Bubble sort failed on empty list.");
    mu_assert(is_sorted(words), "Words should be sorted if empty
.");

    List_destroy(words);

    return NULL;
}

char *test_merge_sort()
{
    List *words = create_words();

    // should work on a list that needs sorting
    List *res = List_merge_sort(words, (List_compare)strcmp);
    mu_assert(is_sorted(res), "Words are not sorted after merge
sort.");

    List *res2 = List_merge_sort(res, (List_compare)strcmp);
    mu_assert(is_sorted(res), "Should still be sorted after merg
e sort.");
    List_destroy(res2);
    List_destroy(res);

    List_destroy(words);
    return NULL;
}


char *all_tests()
{
    mu_suite_start();

    mu_run_test(test_bubble_sort);
    mu_run_test(test_merge_sort);

    return NULL;
}

RUN_TESTS(all_tests);
```

建议你从冒泡排序开始，使它正确，之后再测试归并。我所做的就是编写函数原型和主干，让这三个文件能够编译，但不能通过测试。之后你将实现填充进入之后才能够工作。

实现

你作弊了吗？之后的练习中，我只会给你单元测试，并且让自己实现它。对于你来说，不看这段代码知道你自己实现它是一种很好的练习。下面是 `list_algos.c` 和 `list_algos.h` 的代码：

```
#ifndef lcthw_List_algos_h
#define lcthw_List_algos_h

#include <lcthw/list.h>

typedef int (*List_compare)(const void *a, const void *b);

int List_bubble_sort(List *list, List_compare cmp);

List *List_merge_sort(List *list, List_compare cmp);

#endif
```

```
#include <lcthw/list_algos.h>
#include <lcthw/dbg.h>

inline void ListNode_swap(ListNode *a, ListNode *b)
{
    void *temp = a->value;
    a->value = b->value;
    b->value = temp;
}

int List_bubble_sort(List *list, List_compare cmp)
{
    int sorted = 1;

    if(List_count(list) <= 1) {
        return 0;  // already sorted
    }

    do {
        sorted = 1;
        LIST_FOREACH(list, first, next, cur) {
            if(cur->next) {
                if(cmp(cur->value, cur->next->value) > 0) {
                    ListNode_swap(cur, cur->next);
                    sorted = 0;
                }
            }
        }
    } while(!sorted);

    return 0;
}
```

```c
inline List *List_merge(List *left, List *right, List_compare cm
p)
{
    List *result = List_create();
    void *val = NULL;

    while(List_count(left) > 0 || List_count(right) > 0) {
        if(List_count(left) > 0 && List_count(right) > 0) {
            if(cmp(List_first(left), List_first(right)) <= 0) {
                val = List_shift(left);
            } else {
                val = List_shift(right);
            }

            List_push(result, val);
        } else if(List_count(left) > 0) {
            val = List_shift(left);
            List_push(result, val);
        } else if(List_count(right) > 0) {
            val = List_shift(right);
            List_push(result, val);
        }
    }

    return result;
}

List *List_merge_sort(List *list, List_compare cmp)
{
    if(List_count(list) <= 1) {
        return list;
    }

    List *left = List_create();
    List *right = List_create();
    int middle = List_count(list) / 2;

    LIST_FOREACH(list, first, next, cur) {
        if(middle > 0) {
            List_push(left, cur->value);
        } else {
            List_push(right, cur->value);
        }

        middle--;
    }

    List *sort_left = List_merge_sort(left, cmp);
    List *sort_right = List_merge_sort(right, cmp);

    if(sort_left != left) List_destroy(left);
    if(sort_right != right) List_destroy(right);
```

```c
    return List_merge(sort_left, sort_right, cmp);
}
```

冒泡排序并不难以理解，虽然它非常慢。归并排序更为复杂，实话讲如果我想要牺牲可读性的话，我会花一点时间来优化代码。

归并排序有另一种“自底向上”的实现方式，但是它太难了，我就没有选择它。就像我刚才说的那样，在链表上编写排序算法没有什么意思。你可以把时间都花在使它更快，它比起其他可排序的数据结构会相当版。链表的本质决定了如果你需要对数据进行排序，你就不要使用它们（尤其是单向的）。

你会看到什么

如果一切都正常工作，你会看到这些：

```
$ make clean all
rm -rf build src/lcthw/list.o src/lcthw/list_algos.o tests/list_
algos_tests tests/list_tests
rm -f tests/tests.log
find . -name "*.gc*" -exec rm {} \;
rm -rf `find . -name "*.dSYM" -print`
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  -fPIC   -c -o
src/lcthw/list.o src/lcthw/list.c
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  -fPIC   -c -o
src/lcthw/list_algos.o src/lcthw/list_algos.c
ar rcs build/liblcthw.a src/lcthw/list.o src/lcthw/list_algos.o
ranlib build/liblcthw.a
cc -shared -o build/liblcthw.so src/lcthw/list.o src/lcthw/list_
algos.o
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  build/liblcthw
.a    tests/list_algos_tests.c   -o tests/list_algos_tests
cc -g -O2 -Wall -Wextra -Isrc -rdynamic -DNDEBUG  build/liblcthw
.a    tests/list_tests.c   -o tests/list_tests
sh ./tests/runtests.sh
Running unit tests:
----
RUNNING: ./tests/list_algos_tests
ALL TESTS PASSED
Tests run: 2
tests/list_algos_tests PASS
----
RUNNING: ./tests/list_tests
ALL TESTS PASSED
Tests run: 6
tests/list_tests PASS
$
```

这个练习之后我就不会向你展示这样的输出了，除非有必要向你展示它的工作原理。你应该能知道我运行了测试，并且通过了所有测试。

如何改进

退回去查看算法描述，有一些方法可用于改进这些实现，其中一些是很显然的：

- 归并排序做了大量的链表复制和创建操作，寻找减少它们的办法。
- 归并排序的维基百科描述提到了一些优化，实现它们。
- 你能使用`List_split`和`List_join`（如果你实现了的话）来改进归并排序嘛？
- 浏览所有防御性编程原则，检查并提升这一实现的健壮性，避免`NULL`指针，并且创建一个可选的调试级别的不变量，在排序后实现`is_sorted`的功能。

附加题

- 创建单元测试来比较这两个算法的性能。你需要`man 3 time`来查询基本的时间函数，并且需要运行足够的迭代次数，至少以几秒钟作为样本。
- 改变需要排序的链表中的数据总量，看看耗时如何变化。
- 寻找方法来创建不同长度的随机链表，并且测量需要多少时间，之后将它可视化并与算法的描述对比。
- 尝试解释为什么对链表排序十分麻烦。
- 实现`List_insert_sorted`（有序链表），它使用`List_compare`，接收一个值，将其插入到正确的位置，使链表有序。它与创建链表后再进行排序相比怎么样？
- 尝试实现维基百科上“自底向上”的归并排序。上面的代码已经是C写的了，所以很容易重新创建，但是要试着理解它的工作原理，并与这里的低效版本对比。

练习34：动态数组

> 原文：Exercise 34: Dynamic Array
>
> 译者：飞龙

动态数组是自增长的数组，它与链表有很多相同的特性。它通常占据更少的空间，跑得更快，还有一些其它的优势属性。这个练习会涉及到它的一些缺点，比如从开头移除元素会很慢，并给出解决方案（只从末尾移除）。

动态数组简单地实现为 `void **` 指针的数组，它是预分配内存的，并且指向数据。在链表中你创建了完整的结构体来储存 `void *value` 指针，但是动态数组中你只需要一个储存它们的单个数组。也就是说，你并不需要创建任何其它的指针储存上一个或下一个元素。它们可以直接索引。

我会给你头文件作为起始，你需要为实现打下它们：

```c
#ifndef _DArray_h
#define _DArray_h
#include <stdlib.h>
#include <assert.h>
#include <lcthw/dbg.h>

typedef struct DArray {
    int end;
    int max;
    size_t element_size;
    size_t expand_rate;
    void **contents;
} DArray;

DArray *DArray_create(size_t element_size, size_t initial_max);

void DArray_destroy(DArray *array);

void DArray_clear(DArray *array);

int DArray_expand(DArray *array);

int DArray_contract(DArray *array);

int DArray_push(DArray *array, void *el);

void *DArray_pop(DArray *array);

void DArray_clear_destroy(DArray *array);

#define DArray_last(A) ((A)->contents[(A)->end - 1])
#define DArray_first(A) ((A)->contents[0])
```

```
#define DArray_end(A) ((A)->end)
#define DArray_count(A) DArray_end(A)
#define DArray_max(A) ((A)->max)

#define DEFAULT_EXPAND_RATE 300


static inline void DArray_set(DArray *array, int i, void *el)
{
    check(i < array->max, "darray attempt to set past max");
    if(i > array->end) array->end = i;
    array->contents[i] = el;
error:
    return;
}

static inline void *DArray_get(DArray *array, int i)
{
    check(i < array->max, "darray attempt to get past max");
    return array->contents[i];
error:
    return NULL;
}

static inline void *DArray_remove(DArray *array, int i)
{
    void *el = array->contents[i];

    array->contents[i] = NULL;

    return el;
}

static inline void *DArray_new(DArray *array)
{
    check(array->element_size > 0, "Can't use DArray_new on 0 si
ze darrays.");

    return calloc(1, array->element_size);

error:
    return NULL;
}

#define DArray_free(E) free((E))

#endif
```

这个头文件向你展示了 `static inline` 的新技巧，它就类似 `#define` 宏的工作方式，但是它们更清楚，并且易于编写。如果你需要创建一块代码作为宏，并且不需要代码生成，可以使用 `static inline` 函数。

为链表生成`for`循环的`LIST_FOREACH`不可能写为`static inline`函数，因为它需要生成循环的内部代码块。实现它的唯一方式是灰调函数，但是这不够块，并且难以使用。

之后我会修改代码，并且让你创建`DArray`的单元测试。

```
#include "minunit.h"
#include <lcthw/darray.h>

static DArray *array = NULL;
static int *val1 = NULL;
static int *val2 = NULL;

char *test_create()
{
    array = DArray_create(sizeof(int), 100);
    mu_assert(array != NULL, "DArray_create failed.");
    mu_assert(array->contents != NULL, "contents are wrong in da
rray");
    mu_assert(array->end == 0, "end isn't at the right spot");
    mu_assert(array->element_size == sizeof(int), "element size
is wrong.");
    mu_assert(array->max == 100, "wrong max length on initial si
ze");

    return NULL;
}

char *test_destroy()
{
    DArray_destroy(array);

    return NULL;
}

char *test_new()
{
    val1 = DArray_new(array);
    mu_assert(val1 != NULL, "failed to make a new element");

    val2 = DArray_new(array);
    mu_assert(val2 != NULL, "failed to make a new element");

    return NULL;
}

char *test_set()
{
    DArray_set(array, 0, val1);
    DArray_set(array, 1, val2);

    return NULL;
```

```
}

char *test_get()
{
    mu_assert(DArray_get(array, 0) == val1, "Wrong first value."
);
    mu_assert(DArray_get(array, 1) == val2, "Wrong second value."
);

    return NULL;
}

char *test_remove()
{
    int *val_check = DArray_remove(array, 0);
    mu_assert(val_check != NULL, "Should not get NULL.");
    mu_assert(*val_check == *val1, "Should get the first value."
);
    mu_assert(DArray_get(array, 0) == NULL, "Should be gone.");
    DArray_free(val_check);

    val_check = DArray_remove(array, 1);
    mu_assert(val_check != NULL, "Should not get NULL.");
    mu_assert(*val_check == *val2, "Should get the first value."
);
    mu_assert(DArray_get(array, 1) == NULL, "Should be gone.");
    DArray_free(val_check);

    return NULL;
}

char *test_expand_contract()
{
    int old_max = array->max;
    DArray_expand(array);
    mu_assert((unsigned int)array->max == old_max + array->expan
d_rate, "Wrong size after expand.");

    DArray_contract(array);
    mu_assert((unsigned int)array->max == array->expand_rate + 1
, "Should stay at the expand_rate at least.");

    DArray_contract(array);
    mu_assert((unsigned int)array->max == array->expand_rate + 1
, "Should stay at the expand_rate at least.");

    return NULL;
}

char *test_push_pop()
{
    int i = 0;
    for(i = 0; i < 1000; i++) {
```

```
        int *val = DArray_new(array);
        *val = i * 333;
        DArray_push(array, val);
    }

    mu_assert(array->max == 1201, "Wrong max size.");

    for(i = 999; i >= 0; i--) {
        int *val = DArray_pop(array);
        mu_assert(val != NULL, "Shouldn't get a NULL.");
        mu_assert(*val == i * 333, "Wrong value.");
        DArray_free(val);
    }

    return NULL;
}


char * all_tests() {
    mu_suite_start();

    mu_run_test(test_create);
    mu_run_test(test_new);
    mu_run_test(test_set);
    mu_run_test(test_get);
    mu_run_test(test_remove);
    mu_run_test(test_expand_contract);
    mu_run_test(test_push_pop);
    mu_run_test(test_destroy);

    return NULL;
}

RUN_TESTS(all_tests);
```

这向你展示了所有操作都如何使用，它会使 `DArray` 的实现变得容易：

```
#include <lcthw/darray.h>
#include <assert.h>


DArray *DArray_create(size_t element_size, size_t initial_max)
{
    DArray *array = malloc(sizeof(DArray));
    check_mem(array);
    array->max = initial_max;
    check(array->max > 0, "You must set an initial_max > 0.");

    array->contents = calloc(initial_max, sizeof(void *));
    check_mem(array->contents);
```

```
    array->end = 0;
    array->element_size = element_size;
    array->expand_rate = DEFAULT_EXPAND_RATE;

    return array;

error:
    if(array) free(array);
    return NULL;
}

void DArray_clear(DArray *array)
{
    int i = 0;
    if(array->element_size > 0) {
        for(i = 0; i < array->max; i++) {
            if(array->contents[i] != NULL) {
                free(array->contents[i]);
            }
        }
    }
}

static inline int DArray_resize(DArray *array, size_t newsize)
{
    array->max = newsize;
    check(array->max > 0, "The newsize must be > 0.");

    void *contents = realloc(array->contents, array->max * sizeo
f(void *));
    // check contents and assume realloc doesn't harm the origin
al on error

    check_mem(contents);

    array->contents = contents;

    return 0;
error:
    return -1;
}

int DArray_expand(DArray *array)
{
    size_t old_max = array->max;
    check(DArray_resize(array, array->max + array->expand_rate)
== 0,
            "Failed to expand array to new size: %d",
            array->max + (int)array->expand_rate);

    memset(array->contents + old_max, 0, array->expand_rate + 1)
;
```

```
    return 0;

error:
    return -1;
}

int DArray_contract(DArray *array)
{
    int new_size = array->end < (int)array->expand_rate ? (int)a
rray->expand_rate : array->end;

    return DArray_resize(array, new_size + 1);
}


void DArray_destroy(DArray *array)
{
    if(array) {
        if(array->contents) free(array->contents);
        free(array);
    }
}

void DArray_clear_destroy(DArray *array)
{
    DArray_clear(array);
    DArray_destroy(array);
}

int DArray_push(DArray *array, void *el)
{
    array->contents[array->end] = el;
    array->end++;

    if(DArray_end(array) >= DArray_max(array)) {
        return DArray_expand(array);
    } else {
        return 0;
    }
}

void *DArray_pop(DArray *array)
{
    check(array->end - 1 >= 0, "Attempt to pop from empty array.
");

    void *el = DArray_remove(array, array->end - 1);
    array->end--;

    if(DArray_end(array) > (int)array->expand_rate && DArray_end
(array) % array->expand_rate) {
        DArray_contract(array);
    }
```

```
    return el;
error:
    return NULL;
}
```

这占你展示了另一种处理复杂代码的方法，观察头文件并阅读单元测试，而不是一头扎进 `.c` 实现中。这种“具体的抽象”让你理解代码如何一起工作，并且更容易记住。

优点和缺点

`DArray` 在你需要这些操作时占优势。

- 迭代。你可以仅仅使用基本的 `for` 循环，使用 `DArray_count` 和 `DArray_get` 来完成任务。不需要任何特殊的宏。并且由于不处理指针，它非常快。
- 索引。你可以使用 `DArray_get` 和 `DArray_set` 来随机访问任何元素，但是 `List` 上你就必须经过第N个元素来访问第N+1个元素。
- 销毁。你只需要以两个操作销毁结构体和 `content` 。但是 `List` 需要一些列的 `free` 调用同时遍历每个元素。
- 克隆。你只需要复制结构体和 `content` ，用两步复制整个结构。 `List` 需要遍历所有元素并且复制每个 `ListNode` 和值。
- 排序。你已经见过了，如果你需要对数据排序， `List` 非常麻烦。 `DArray` 上可以实现所有高效的排序算法，因为你可以随机访问任何元素。
- 大量数据。如果你需要储存大量数据， `DArray` 由于基于 `content` ，比起相同数量的 `ListNode` 占用更少空间而占优。

然而 `List` 在这些操作上占优势。

- 在开头插入和移除元素。 `DArray` 需要特殊的优化来高效地完成它，并且通常还需要一些复制操作。
- 分割和连接。 `List` 只需要复制一些指针就能完成，但是 `DArray` 需要复制涉及到的所有数组。
- 少量数据。如果你只需要存储几个元素，通常使用 `List` 所需的空间要少于 `DArray` ，因为 `DArray` 需要考虑到日后的添加而扩展背后的空间，但是 `List` 只需要元素所需的空间。

考虑到这些，我更倾向使用 `DArray` 来完成其它人使用 `List` 所做的大部分事情。对于任何需要少量节点并且在两端插入删除的，我会使用 `List` 。我会想你展示两个相似的数据结构，叫做 `Stack` 和 `Queue` ，它们也很重要。

如何改进

像往常一样，浏览每个函数和操作，并且执行防御性编程检查，以及添加先决条件、不变量等任何可以使实现更健壮的东西。

附加题

- 改进单元测试来覆盖耕作操作，并使用 `for` 循环来测试迭代。
- 研究 `DArray` 上如何实现冒泡排序和归并排序，但是不要马上实现它们。我会在下一张实现 `DArray` 的算法，之后你可以完成它。
- 为一些常用的操作编写一些性能测试，并与 `List` 中的相同操作比较。你已经做过很多次了，但是这次需要编写重复执行所涉及操作的单元测试，之后在主运行器中计时。
- 观察 `DArray_expand` 如何使用固定增长（ `size + 300` ）来实现。通常动态数组都以倍数增长（ `size * 2` ）的方式实现，但是我发现它会花费无用的内存并且没有真正取得性能收益。测试我的断言，并且看看什么情况下需要倍数增长而不是固定增长。

练习35：排序和搜索

> 原文：Exercise 35: Sorting And Searching
>
> 译者：飞龙

这个练习中我打算涉及到四个排序算法和一个搜索算法。排序算法是快速排序、堆排序、归并排序和基数排序。之后在你完成基数排序之后，我打算想你展示二分搜索。

然而，我是一个懒人，大多数C标准库都实现了堆排序、快速排序和归并排序算法，你可以直接使用它们：

```c
#include <lcthw/darray_algos.h>
#include <stdlib.h>

int DArray_qsort(DArray *array, DArray_compare cmp)
{
    qsort(array->contents, DArray_count(array), sizeof(void *), cmp);
    return 0;
}

int DArray_heapsort(DArray *array, DArray_compare cmp)
{
    return heapsort(array->contents, DArray_count(array), sizeof(void *), cmp);
}

int DArray_mergesort(DArray *array, DArray_compare cmp)
{
    return mergesort(array->contents, DArray_count(array), sizeof(void *), cmp);
}
```

这就是 `darray_algos.c` 文件的整个实现，它在大多数现代Unix系统上都能运行。它们的每一个都使用 `DArray_compare` 对 `contents` 中储存的无类型指针进行排序。我也要向你展示这个头文件：

```c
#ifndef darray_algos_h
#define darray_algos_h

#include <lcthw/darray.h>

typedef int (*DArray_compare)(const void *a, const void *b);

int DArray_qsort(DArray *array, DArray_compare cmp);

int DArray_heapsort(DArray *array, DArray_compare cmp);

int DArray_mergesort(DArray *array, DArray_compare cmp);

#endif
```

大小几乎一样，你也应该能预料到。接下来你可以了解单元测试中这三个函数如何使用：

```c
#include "minunit.h"
#include <lcthw/darray_algos.h>

int testcmp(char **a, char **b)
{
    return strcmp(*a, *b);
}

DArray *create_words()
{
    DArray *result = DArray_create(0, 5);
    char *words[] = {"asdfasfd", "werwar", "13234", "asdfasfd",
"oioj"};
    int i = 0;

    for(i = 0; i < 5; i++) {
        DArray_push(result, words[i]);
    }

    return result;
}

int is_sorted(DArray *array)
{
    int i = 0;

    for(i = 0; i < DArray_count(array) - 1; i++) {
        if(strcmp(DArray_get(array, i), DArray_get(array, i+1))
> 0) {
            return 0;
        }
    }
```

```
    return 1;
}

char *run_sort_test(int (*func)(DArray *, DArray_compare), const
 char *name)
{
    DArray *words = create_words();
    mu_assert(!is_sorted(words), "Words should start not sorted.
");

    debug("--- Testing %s sorting algorithm", name);
    int rc = func(words, (DArray_compare)testcmp);
    mu_assert(rc == 0, "sort failed");
    mu_assert(is_sorted(words), "didn't sort it");

    DArray_destroy(words);

    return NULL;
}

char *test_qsort()
{
    return run_sort_test(DArray_qsort, "qsort");
}

char *test_heapsort()
{
    return run_sort_test(DArray_heapsort, "heapsort");
}

char *test_mergesort()
{
    return run_sort_test(DArray_mergesort, "mergesort");
}


char * all_tests()
{
    mu_suite_start();

    mu_run_test(test_qsort);
    mu_run_test(test_heapsort);
    mu_run_test(test_mergesort);

    return NULL;
}

RUN_TESTS(all_tests);
```

你需要注意的事情是第四行`testcmp`的定义，它困扰了我一整天。你必须使用`char **`而不是`char *`，因为`qsort`会向你提供指向`content`数组中指针的指针。原因是`qsort`会打扫数组，使用你的比较函数来处理数组中每个元素的指针。因为我在`contents`中存储指针，所以你需要使用指针的指针。

有了这些之后，你只需要实现三个困难的搜索算法，每个大约20行。你应该在这里停下来，不过这本书的一部分就是学习这些算法的原理，附加题会涉及到实现这些算法。

基数排序和二分搜索

既然你打算自己实现快速排序、堆排序和归并排序，我打算向你展示一个流行的算法叫做基数排序。它的实用性很小，只能用于整数数组，并且看上去像魔法一样。这里我打算常见一个特殊的数据结构，叫做`RadixMap`，用于将一个整数映射为另一个。

下面是为新算法创建的头文件，其中也含有数据结构：

```c
#ifndef _radixmap_h
#include <stdint.h>

typedef union RMElement {
    uint64_t raw;
    struct {
        uint32_t key;
        uint32_t value;
    } data;
} RMElement;

typedef struct RadixMap {
    size_t max;
    size_t end;
    uint32_t counter;
    RMElement *contents;
    RMElement *temp;
} RadixMap;


RadixMap *RadixMap_create(size_t max);

void RadixMap_destroy(RadixMap *map);

void RadixMap_sort(RadixMap *map);

RMElement *RadixMap_find(RadixMap *map, uint32_t key);

int RadixMap_add(RadixMap *map, uint32_t key, uint32_t value);

int RadixMap_delete(RadixMap *map, RMElement *el);

#endif
```

你看到了其中有许多和 `Dynamic Array` 或 `List` 数据结构相同的操作，不同就在于我只处理固定32位大小的 `uint32_t` 正忽视。我也会想你介绍C语言的一个新概念，叫做 `union` 。

C联合体

联合体是使用不同方式引用内存中同一块区域的方法。它们的工作方式，就像你把它定义为 `sturct` ，然而，每个元素共享同一片内存区域。你可以认为，联合体是内存中的一幅画，所有颜色不同的元素都重叠在它上面。

它可以用于节约内存，或在不同格式之间转换内存块。它的第一个用途就是实现“可变类型”，你可以创建一个带有类型“标签”的结构体，之后在其中创建含有多种类型的联合体。用于在内存的不同格式之间转换时，只需要定义两个结构体，访问正确的那个类型。

首先让我向你展示如何使用C联合体构造可变类型：

```
#include <stdio.h>

typedef enum {
    TYPE_INT,
    TYPE_FLOAT,
    TYPE_STRING,
} VariantType;

struct Variant {
    VariantType type;
    union {
        int as_integer;
        float as_float;
        char *as_string;
    } data;
};

typedef struct Variant Variant;

void Variant_print(Variant *var)
{
    switch(var->type) {
        case TYPE_INT:
           printf("INT: %d\n", var->data.as_integer);
           break;
        case TYPE_FLOAT:
           printf("FLOAT: %f\n", var->data.as_float);
           break;
        case TYPE_STRING:
           printf("STRING: %s\n", var->data.as_string);
           break;
        default:
           printf("UNKNOWN TYPE: %d", var->type);
    }
}

int main(int argc, char *argv[])
{
    Variant a_int = {.type = TYPE_INT, .data.as_integer = 100};
    Variant a_float = {.type = TYPE_FLOAT, .data.as_float = 100.
34};
    Variant a_string = {.type = TYPE_STRING, .data.as_string = "
YO DUDE!"};

    Variant_print(&a_int);
    Variant_print(&a_float);
    Variant_print(&a_string);

    // here's how you access them
    a_int.data.as_integer = 200;
```

```c
    a_float.data.as_float = 2.345;
    a_string.data.as_string = "Hi there.";

    Variant_print(&a_int);
    Variant_print(&a_float);
    Variant_print(&a_string);

    return 0;
}
```

你可以在许多动态语言实现中发现它。对于为语言中所有基本类型，代码中首先定义了一些带有变迁的可变类型，之后通常给你所创建的类型打上 `object` 标签。这样的好处就是 `Variant` 通常只需要 `VariantType type` 标签的空间，加上联合体最大成员的空间，因为C将 `Variant.data` 的每个元素堆起来，它们是重叠的，只保证有足够的空间放下最大的元素。

`radixmap.h` 文件中我创建了 `RMElement` 联合体，用于在类型之间转换内存块。这里，我希望存储 `uint64_t` 定长整数用于排序目录，但是我也希望使用两个 `uint32_t` 用于表示数据的 `key` 和 `value` 对。通过使用联合体我就能够使用所需的两种不同方法来访问内存。

实现

接下来是实际的 `RadixMap` 对于这些操作的实现：

```c
/*
* Based on code by Andre Reinald then heavily modified by Zed A.
 Shaw.
*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <lcthw/radixmap.h>
#include <lcthw/dbg.h>

RadixMap *RadixMap_create(size_t max)
{
    RadixMap *map = calloc(sizeof(RadixMap), 1);
    check_mem(map);

    map->contents = calloc(sizeof(RMElement), max + 1);
    check_mem(map->contents);

    map->temp = calloc(sizeof(RMElement), max + 1);
    check_mem(map->temp);

    map->max = max;
    map->end = 0;
```

```
    return map;
error:
    return NULL;
}

void RadixMap_destroy(RadixMap *map)
{
    if(map) {
        free(map->contents);
        free(map->temp);
        free(map);
    }
}


#define ByteOf(x,y) (((uint8_t *)x)[(y)])

static inline void radix_sort(short offset, uint64_t max, uint64
_t *source, uint64_t *dest)
{
    uint64_t count[256] = {0};
    uint64_t *cp = NULL;
    uint64_t *sp = NULL;
    uint64_t *end = NULL;
    uint64_t s = 0;
    uint64_t c = 0;

    // count occurences of every byte value
    for (sp = source, end = source + max; sp < end; sp++) {
        count[ByteOf(sp, offset)]++;
    }

    // transform count into index by summing elements and storin
g into same array
    for (s = 0, cp = count, end = count + 256; cp < end; cp++) {
        c = *cp;
        *cp = s;
        s += c;
    }

    // fill dest with the right values in the right place
    for (sp = source, end = source + max; sp < end; sp++) {
        cp = count + ByteOf(sp, offset);
        dest[*cp] = *sp;
        ++(*cp);
    }
}

void RadixMap_sort(RadixMap *map)
{
    uint64_t *source = &map->contents[0].raw;
    uint64_t *temp = &map->temp[0].raw;
```

```
    radix_sort(0, map->end, source, temp);
    radix_sort(1, map->end, temp, source);
    radix_sort(2, map->end, source, temp);
    radix_sort(3, map->end, temp, source);
}

RMElement *RadixMap_find(RadixMap *map, uint32_t to_find)
{
    int low = 0;
    int high = map->end - 1;
    RMElement *data = map->contents;

    while (low <= high) {
        int middle = low + (high - low)/2;
        uint32_t key = data[middle].data.key;

        if (to_find < key) {
            high = middle - 1;
        } else if (to_find > key) {
            low = middle + 1;
        } else {
            return &data[middle];
        }
    }

    return NULL;
}

int RadixMap_add(RadixMap *map, uint32_t key, uint32_t value)
{
    check(key < UINT32_MAX, "Key can't be equal to UINT32_MAX.")
;

    RMElement element = {.data = {.key = key, .value = value}};
    check(map->end + 1 < map->max, "RadixMap is full.");

    map->contents[map->end++] = element;

    RadixMap_sort(map);

    return 0;

error:
    return -1;
}

int RadixMap_delete(RadixMap *map, RMElement *el)
{
    check(map->end > 0, "There is nothing to delete.");
    check(el != NULL, "Can't delete a NULL element.");

    el->data.key = UINT32_MAX;
```

```c
    if(map->end > 1) {
        // don't bother resorting a map of 1 length
        RadixMap_sort(map);
    }

    map->end--;

    return 0;
error:
    return -1;
}
```

像往常一样键入它并使它通过单元测试，之后我会解释它。尤其要注意 `radix_sort` 函数，我实现它的方法非常特别。

```c
#include "minunit.h"
#include <lcthw/radixmap.h>
#include <time.h>

static int make_random(RadixMap *map)
{
    size_t i = 0;

    for (i = 0; i < map->max - 1; i++) {
        uint32_t key = (uint32_t)(rand() | (rand() << 16));
        check(RadixMap_add(map, key, i) == 0, "Failed to add key
 %u.", key);
    }

    return i;

error:
    return 0;
}

static int check_order(RadixMap *map)
{
    RMElement d1, d2;
    unsigned int i = 0;

    // only signal errors if any (should not be)
    for (i = 0; map->end > 0 && i < map->end-1; i++) {
        d1 = map->contents[i];
        d2 = map->contents[i+1];

        if(d1.data.key > d2.data.key) {
            debug("FAIL:i=%u, key: %u, value: %u, equals max? %d
\n", i, d1.data.key, d1.data.value,
                        d2.data.key == UINT32_MAX);
            return 0;
```

```
        }
    }

    return 1;
}

static int test_search(RadixMap *map)
{
    unsigned i = 0;
    RMElement *d = NULL;
    RMElement *found = NULL;

    for(i = map->end / 2; i < map->end; i++) {
        d = &map->contents[i];
        found = RadixMap_find(map, d->data.key);
        check(found != NULL, "Didn't find %u at %u.", d->data.ke
y, i);
        check(found->data.key == d->data.key, "Got the wrong res
ult: %p:%u looking for %u at %u",
                found, found->data.key, d->data.key, i);
    }

    return 1;
error:
    return 0;
}

// test for big number of elements
static char *test_operations()
{
    size_t N = 200;

    RadixMap *map = RadixMap_create(N);
    mu_assert(map != NULL, "Failed to make the map.");
    mu_assert(make_random(map), "Didn't make a random fake radix
 map.");

    RadixMap_sort(map);
    mu_assert(check_order(map), "Failed to properly sort the Rad
ixMap.");

    mu_assert(test_search(map), "Failed the search test.");
    mu_assert(check_order(map), "RadixMap didn't stay sorted aft
er search.");

    while(map->end > 0) {
        RMElement *el = RadixMap_find(map, map->contents[map->en
d / 2].data.key);
        mu_assert(el != NULL, "Should get a result.");

        size_t old_end = map->end;

        mu_assert(RadixMap_delete(map, el) == 0, "Didn't delete
```

```
it.");
        mu_assert(old_end - 1 == map->end, "Wrong size after del
ete.");

        // test that the end is now the old value, but uint32 ma
x so it trails off
        mu_assert(check_order(map), "RadixMap didn't stay sorted
 after delete.");
    }

    RadixMap_destroy(map);

    return NULL;
}


char *all_tests()
{
    mu_suite_start();
    srand(time(NULL));

    mu_run_test(test_operations);

    return NULL;
}

RUN_TESTS(all_tests);
```

我不应该向你解释关于测试的过多东西，它只是模拟将随机正是放入 `RadixMap` ，确保你可以可靠地将其取出。也不是非常有趣。

在 `radixmap.c` 中的大多数操作都易于理解，如果你阅读代码的话。下面是每个基本函数作用及其工作原理的描述：

RadixMap_create

像往常一样，我分配了结构体所需的内存，结构体在 `radixmap.h` 中定义。当后面涉及到 `radix_sort` 时我会使用 `temp` 和 `contents` 。

RadixMap_destroy

同样，销毁我所创建的东西。

radix_sort

这个数据结构的灵魂，我会在下一节中解释其作用。

RadixMap_sort

它使用了 `radix_sort` 函数来实际对 `contents` 进行排序。

RadixMap_find

使用二分搜索算法来寻找提供的`key`，我之后会解释它的原理。

RadixMap_add

使用`RadixMap_sort`函数，它会在末尾添加`key`和`value`，然后简单地重新排序使一切元素都有序。一旦排序完，`RadixMap_find`会正确工作，因为它是二分搜索。

RadixMap_delete

工作方式类似`RadixMap_add`，除了“删除”结构中的元素，通过将它们的值设为无符号的32为整数的最大值，也就是`UINT32_MAX`。这意味着你不能使用这个值作为合法的键，但是它是元素删除变得容易。简单设置它之后排序，它会被移动到末尾，这就算删除了。

学习我所描述的代码，接下来还剩`RadixMap_sort`，`radix_sort`和`RadixMap_find`需要了解。

RadixMap_find 和二分搜索

我首先以二分搜索如何实现开始。二分搜索是一种简单算法，大多数人都可以直观地理解。实际上，你可以取一叠游戏卡片（或带有数字的卡片）来手动操作。下面是该函数的工作方式，也是二分搜索的原理：

- 基于数组大小设置上界和下界。
- 获取上下界之间的中间元素。
- 如果键小于这个元素的值，就一定在它前面，所以上界设置为中间元素。
- 如果键大于这个元素的值，就一定在它后面，所以下界设置为中间元素。
- 继续循环直到上界和下界越过了彼此。如果退出了循环则没有找到。

你实际上所做的事情是，通过挑选中间的值来比较，猜出`key`可能的位置。由于数据是有序的，你知道`key`一定会在它前面或者后面，这样就能把搜索区域分成两半。之后你继续搜索知道找到他，或者越过了边界并穷尽了搜索空间。

RadixMap_sort 和 radix_sort

如果你事先手动模拟基数排序，它就很易于理解。这个算法利用了一个现象，数字都以十进制字符的序列来表示，按照“不重要”到“重要”的顺序排列。之后它通过十进制字符来选取数字并且将它们储存在桶中，当它处理完所有字符时，数字就排好序了。一开始它看上去像是魔法，浏览代码也的确如此，但是你要尝试手动执行它。

为了解释这个算法，需要先写下一组三位的十进制数，以随机的顺序，假设就是223、912、275、100、633、120 和 380。

- 按照它们的个位，将数字放入桶中：`[380, 100, 120], [912], [633, 223], [275]`。
- 现在遍历每个桶中的数字，接着按十位排序：`[100], [912], [120, 223], [633], [275], [380]`。

- 现在每个桶都包含了按照个位和十位排序后的数字。接着我需要按照这个顺序遍历，并把它们放入最后百位的桶中： `[100, 120], [223, 275], [380], [633], [912]` 。
- 到现在为止，每个数字都按照百位、十位和个位排序，并且如果我按照顺序遍历每个桶，我会得到最终排序的结果： `100, 120, 223, 275, 380, 633, 912` 。

确保你多次重复了这个过程，便于你理解它如何工作。这实在是一种机智的算法，并且最重要的是它对于任何大小的数字都有效。所以你可以用它来排序比较大的数字，因为你一次只是处理一位。

在我的环境下，“字符”是独立的8位字节，所以我需要256个桶来储存这些数字按照字节的分布结果。我需要一种方法来储存它，并且不需要花费太多的空间。如果你查看 `radix_sort` ，首先我会构建 `count` 直方图，便于我了解对于给定的 `offset` ，每个字节的频率。

一旦我知道了每一种字节的数量（共有256种），我就可以将目标数组用于存储这些值的分布。比如，如果0x00的数量为10个，我就可以将它们放在目标数组的前10个位置中。这可以让我索引到它们在目标数组中的位置，这就是 `radix_sort` 中的第二个 `for` 循环。

最后，当我知道它们在目标数组中储存在哪里，我只是遍历 `source` 数组对于当前 `offset` 的所有字节，并且将数值按顺序放入它们的位置中。 `ByteOf` 宏的使用有助于保持代码整洁，因为它需要一些指针的黑魔法，但是最后当 `for` 循环结束之后，所有整数都会按照它们的字节放入桶中。

我在 `RadixMap_sort` 中对这些64位的整数按照它们的前32位进行排序，这非常有意思。还记得我是如何将键和值放入 `RMElement` 类型的联合体了吗？这意味着如果要按照键来对这个数组排序，我只需要对每个整数前4个字节（32位/8位每字节）进行排序。

如果你观察 `RadixMap_sort` ，你会看到我获取了 `contents` 和 `temp` 的便利指针，用于源数组和目标数组，之后我四次调用 `radix_sort` 。每次调用我将源数组和目标数组替换为下一字节的情况。当我完成时， `radix_sort` 就完成了任务，并且 `contents` 中也有了最后的结果。

如何改进

这个实现有个很大的缺点，就是它遍历了整个数组四次。它执行地很快，但是如果你通过需要排序的数值大小来限制排序的总量，会更好一些。

有两个方法可以用于改进这个实现：

- 使用二分搜索来寻找新元素的最小位置，只对这个位置到微末之间进行排序。你需要找到它，将新元素放到末尾，之后对它们之间进行排序。大多数情况下这会显著地缩减排序范围。
- 跟踪当前所使用的最大的键，之后只对足够的位数进行排序，来处理这个键。你也可以跟踪最小的数值，之后只对范围中必要的字节进行排序。为了这样做，你需要关心CPU的整数存储顺序（大小端序）。

附加题

+ 实现快速排序、堆排序和归并排序，并且提供一个 `#define` 让其他人在二者（标准库和你的实现）当中进行选择，或者创建另一套不同名称的函数。使用我教给你的技巧，阅读维基百科的算法页面，之后参照伪代码来实现它。
+ 对比你的实现和标准库实现的性能。
+ 使用这些排序函数创建 `DArray_sort_add` ，它可以向 `DArray` 添加元素，但是随后对数组排序。
+ 编写 `DArray_find` ，使用 `RadixMap_find` 中的二分搜索算法和 `DArray_compare` ，来在有序的 `DArray` 中寻找元素。

练习36：更安全的字符串

> 原文：Exercise 36: Safer Strings
>
> 译者：飞龙

我已经在练习26中，构建 `devpkg` 的时候介绍了Better String库。这个练习让你从现在开始熟悉 `bstring` 库，并且明白C风格字符串为什么十分糟糕。之后你需要修改 `liblcthw` 的代码来使用 `bstring` 。

为什么C风格字符串十分糟糕

当人们谈论C的问题时，“字符串”的概念永远是首要缺陷之一。你已经用过它们，并且我也谈论过它们的种种缺陷，但是对为什么C字符串拥有缺陷，以及为什么一直是这样没有明确的解释。我会试着现在做出解释，部分原因是C风格字符串经过数十年的使用，有足够的证据表明它们是个非常糟糕的东西。

对于给定的任何C风格字符串，都不可能验证它是否有效。

- 以 `'\0'` 结尾的C字符串是有效的。
- 任何处理无效C字符串的循环都是无限的（或者造成缓冲区溢出）。
- C字符串没有确定的长度，所以检查它们的唯一方法就是遍历它来观察循环是否正确终止。
- 所以，不通过有限的循环就不可能验证C字符串。

这个逻辑非常简单。你不能编写一个循环来验证C字符串是否有效，因为无效的字符串导致循环永远不会停止。就是这样，唯一的解决方案就是包含大小。一旦你知道了大小，你可以避免无限循环问题。如果你观察练习27中我向你展示的两个函数：

> 译者注：检验C风格字符串是否有效等价于“停机问题”，这是一个非常著名的不可解问题。

```c
void copy(char to[], char from[])
{
    int i = 0;

    // while loop will not end if from isn't '\0' terminated
    while((to[i] = from[i]) != '\0') {
        ++i;
    }
}

int safercopy(int from_len, char *from, int to_len, char *to)
{
    int i = 0;
    int max = from_len > to_len - 1 ? to_len - 1 : from_len;

    // to_len must have at least 1 byte
    if(from_len < 0 || to_len <= 0) return -1;

    for(i = 0; i < max; i++) {
        to[i] = from[i];
    }

    to[to_len - 1] = '\0';

    return i;
}
```

想象你想要向 `copy` 函数添加检查来确保 `from` 字符串有效。你该怎么做呢？你编写了一个循环来检查字符串是否已 `'\0'` 结尾。哦，等一下，如果字符串不以 `'\0'` 结尾，那它怎么让循环停下？不可能停下，所以无解。

无论你怎么做，你都不能在不知道字符串长度的情况下检查C字符串的有效性，这里 `safercopy` 包含了程度。这个函数没有相同的问题，因为他的循环一定会中止，即使你传入了错误的大小，大小也是有限的。

> 译者注：但是问题来了，对于一个C字符串，你怎么获取其大小？你需要在这个函数之前调用 `strlen` ，又是一个无限循环问题。

于是， `bstring` 库所做的事情就是创建一个结构体，它总是包含字符串长度。由于这个长度对于 `bstring` 来说总是可访问的，它上面的所有操作都会更安全。循环是有限的，内容也是有效的，并且这个主要的缺陷也不存在了。BString库也带有大量所需的字串操作，比如分割、格式化、搜索，并且大多数都会正确并安全地执行。

`bstring` 中也可能有缺陷，但是经过这么长时间，可能性已经很低了。 `glibc` 中也有缺陷，所以你让程序员怎么做才好呢？

使用 bstrlib

有很多改进后的字符串库，但是我最喜欢 `bstrlib` ，因为它只有一个程序集，并且具有大多数所需的字符串功能。你已经在使用它了，所以这个练习中你需要从Better String获取两个文件， `bstrlib.c` 和 `bstrlib.h` 。

下面是我在 `liblcthw` 项目目录里所做的事情：

```
$ mkdir bstrlib
$ cd bstrlib/
$ unzip ~/Downloads/bstrlib-05122010.zip
Archive:  /Users/zedshaw/Downloads/bstrlib-05122010.zip
...
$ ls
bsafe.c             bstraux.c       bstrlib.h       bstrwrap.h
    license.txt     test.cpp
bsafe.h             bstraux.h       bstrlib.txt     cpptest.cpp
    porting.txt     testaux.c
bstest.c    bstrlib.c       bstrwrap.cpp    gpl.txt         secu
rity.txt
$ mv bstrlib.h bstrlib.c ../src/lcthw/
$ cd ../
$ rm -rf bstrlib
# make the edits
$ vim src/lcthw/bstrlib.c
$ make clean all
...
$
```

在第14行你可以看到，我编辑了 `bstrlib.c` 文件，来将它移动到新的位置，并且修复OSX上的bug。下面是差异：

```
25c25
< #include "bstrlib.h"
---
> #include <lcthw/bstrlib.h>
2759c2759
< #ifdef __GNUC__
---
> #if defined(__GNUC__) && !defined(__APPLE__)
```

我把包含修改为 `<lcthw/bstrlib.h>` ，然后修复2759行 `ifdef` 的问题。

学习使用该库

这个练习很短，只是让你准备好剩余的练习，它们会用到这个库。接下来两个联系中，我会使用 `bstrlib.c` 来创建Hashmap`数据结构。

你现在应该阅读头文件和实现，之后编写`tests/bstr_tests.c`来测试下列函数，来熟悉这个库：

`bfromcstr`

从C风格字符串中创建一个`bstring`。

`blk2bstr`

与上面相同，但是可以提供缓冲区长度。

`bstrcpy`

复制`bstring`。

`bassign`

将一个`bstring`赋值为另一个。

`bassigncstr`

将`bsting`的内容设置为C字符串的内容。

`bassignblk`

将`bsting`的内容设置为C字符串的内容，但是可以提供长度。

`bdestroy`

销毁`bstring`。

`bconcat`

在一个`bstring`末尾连接另一个。

`bstricmp`

比较两个`bstring`，返回值与`strcmp`相同。

`biseq`

检查两个`bstring`是否相等。

`binstr`

判断一个`bstring`是否被包含于另一个。

`bfindreplace`

在一个`bstring`中寻找另一个，并且将其替换为别的。

`bsplit`

将`bstring`分割为`bstrList`。

`bformat`

执行字符串格式化，十分便利。

`blength`

获取`bstring`的长度。

`bdata`

获取`bstring`的数据。

`bchar`

获得`bstring`中的字符。

你的测试应该覆盖到所有这些操作，以及你从头文件中发现的更多有趣的东西。在`valgrind`下运行测试，确保内存使用正确。

练习37：哈希表

> 原文：Exercise 37: Hashmaps
>
> 译者：飞龙

哈希表（`HashMap`、`HashTable`以及`Dictionary`）广泛用于许多动态编程语言来储存键值对的数据。哈希表通过在键上执行“哈希”运算产生整数，之后使用它来寻找相应的桶来获取或储存值。它是非常快速的使用数据结构，因为它适用于任何数据并且易于实现。

下面是哈希表（也叫作字典）的一个使用示例：

```
fruit_weights = {'Apples': 10, 'Oranges': 100, 'Grapes': 1.0}

for key, value in fruit_weights.items():
    print key, "=", value
```

几乎所有现代语言都具备这种特性，所以许多人写完代码都不知道它实际上如何工作。通过在C中创建`Hashmap`数据结构，我会向你展示它的工作原理。我会从头文件开始，来谈论整个数据结构。

```
#ifndef _lcthw_Hashmap_h
#define _lcthw_Hashmap_h

#include <stdint.h>
#include <lcthw/darray.h>

#define DEFAULT_NUMBER_OF_BUCKETS 100

typedef int (*Hashmap_compare)(void *a, void *b);
typedef uint32_t (*Hashmap_hash)(void *key);

typedef struct Hashmap {
    DArray *buckets;
    Hashmap_compare compare;
    Hashmap_hash hash;
} Hashmap;

typedef struct HashmapNode {
    void *key;
    void *data;
    uint32_t hash;
} HashmapNode;

typedef int (*Hashmap_traverse_cb)(HashmapNode *node);

Hashmap *Hashmap_create(Hashmap_compare compare, Hashmap_hash);
void Hashmap_destroy(Hashmap *map);

int Hashmap_set(Hashmap *map, void *key, void *data);
void *Hashmap_get(Hashmap *map, void *key);

int Hashmap_traverse(Hashmap *map, Hashmap_traverse_cb traverse_
cb);

void *Hashmap_delete(Hashmap *map, void *key);

#endif
```

这个结构就是 `Hashmap` ，含有许多 `HashmapNode` 节点。观察 `Hashmap` 你会看到它类似这样：

`DArray *buckets`

一个动态数组，设置为100个桶的固定大小。每个桶会含有一个 `DArray` ，来实际存档 `HashmapNode` 对。

`Hashmap_compare compare`

这是一个比较函数，被 `Hashmap` 用于实际用过键寻找元素。它应该和其它的比较函数类似，并且默认设置为 `bstrcmp` 来比较字符串。

`Hashmap_hash`

这是哈希函数，它用于接收键，处理它的内容，之后产生一个 `uint32_t` 索引数值。之后你会看到默认的实现。

这些告诉了你数据如何存储，但是用作 `buckets` 的 `DArray` 还没有创建。要记住它具有二层结构；

- 第一层有100个桶，数据基于它们的哈希值储存在桶中。
- 每个桶都是一个 `DArray` ，其中含有 `HashmapNode` ，添加时只是简单地附加到末尾。

`HashMapNode` 由下面三个元素组成：

`void *key`

键值对的键。

`void *value`

键值对的值。

`uint32_t hash`

计算出的哈希值，它用于使查找该节点更加迅速，只要判断键是否相等。

有文件的剩余部分没有新的东西，所以我现在可以向你展示 `hashmap.c` 的实现了：

```
#undef NDEBUG
#include <stdint.h>
#include <lcthw/hashmap.h>
#include <lcthw/dbg.h>
#include <lcthw/bstrlib.h>

static int default_compare(void *a, void *b)
{
    return bstrcmp((bstring)a, (bstring)b);
}

/**
 * Simple Bob Jenkins's hash algorithm taken from the
 * wikipedia description.
 */
static uint32_t default_hash(void *a)
{
    size_t len = blength((bstring)a);
    char *key = bdata((bstring)a);
    uint32_t hash = 0;
    uint32_t i = 0;

    for(hash = i = 0; i < len; ++i)
    {
```

```
        hash += key[i];
        hash += (hash << 10);
        hash ^= (hash >> 6);
    }

    hash += (hash << 3);
    hash ^= (hash >> 11);
    hash += (hash << 15);

    return hash;
}


Hashmap *Hashmap_create(Hashmap_compare compare, Hashmap_hash ha
sh)
{
    Hashmap *map = calloc(1, sizeof(Hashmap));
    check_mem(map);

    map->compare = compare == NULL ? default_compare : compare;
    map->hash = hash == NULL ? default_hash : hash;
    map->buckets = DArray_create(sizeof(DArray *), DEFAULT_NUMBE
R_OF_BUCKETS);
    map->buckets->end = map->buckets->max; // fake out expanding
 it
    check_mem(map->buckets);

    return map;

error:
    if(map) {
        Hashmap_destroy(map);
    }

    return NULL;
}


void Hashmap_destroy(Hashmap *map)
{
    int i = 0;
    int j = 0;

    if(map) {
        if(map->buckets) {
            for(i = 0; i < DArray_count(map->buckets); i++) {
                DArray *bucket = DArray_get(map->buckets, i);
                if(bucket) {
                    for(j = 0; j < DArray_count(bucket); j++) {
                        free(DArray_get(bucket, j));
                    }
                    DArray_destroy(bucket);
                }
```

```
            }
            DArray_destroy(map->buckets);
        }

        free(map);
    }
}

static inline HashmapNode *Hashmap_node_create(int hash, void *k
ey, void *data)
{
    HashmapNode *node = calloc(1, sizeof(HashmapNode));
    check_mem(node);

    node->key = key;
    node->data = data;
    node->hash = hash;

    return node;

error:
    return NULL;
}


static inline DArray *Hashmap_find_bucket(Hashmap *map, void *ke
y,
        int create, uint32_t *hash_out)
{
    uint32_t hash = map->hash(key);
    int bucket_n = hash % DEFAULT_NUMBER_OF_BUCKETS;
    check(bucket_n >= 0, "Invalid bucket found: %d", bucket_n);
    *hash_out = hash; // store it for the return so the caller c
an use it


    DArray *bucket = DArray_get(map->buckets, bucket_n);

    if(!bucket && create) {
        // new bucket, set it up
        bucket = DArray_create(sizeof(void *), DEFAULT_NUMBER_OF
_BUCKETS);
        check_mem(bucket);
        DArray_set(map->buckets, bucket_n, bucket);
    }

    return bucket;

error:
    return NULL;
}
```

```c
int Hashmap_set(Hashmap *map, void *key, void *data)
{
    uint32_t hash = 0;
    DArray *bucket = Hashmap_find_bucket(map, key, 1, &hash);
    check(bucket, "Error can't create bucket.");

    HashmapNode *node = Hashmap_node_create(hash, key, data);
    check_mem(node);

    DArray_push(bucket, node);

    return 0;

error:
    return -1;
}

static inline int Hashmap_get_node(Hashmap *map, uint32_t hash,
DArray *bucket, void *key)
{
    int i = 0;

    for(i = 0; i < DArray_end(bucket); i++) {
        debug("TRY: %d", i);
        HashmapNode *node = DArray_get(bucket, i);
        if(node->hash == hash && map->compare(node->key, key) ==
0) {
            return i;
        }
    }

    return -1;
}

void *Hashmap_get(Hashmap *map, void *key)
{
    uint32_t hash = 0;
    DArray *bucket = Hashmap_find_bucket(map, key, 0, &hash);
    if(!bucket) return NULL;

    int i = Hashmap_get_node(map, hash, bucket, key);
    if(i == -1) return NULL;

    HashmapNode *node = DArray_get(bucket, i);
    check(node != NULL, "Failed to get node from bucket when it
should exist.");

    return node->data;

error: // fallthrough
    return NULL;
}
```

```c
int Hashmap_traverse(Hashmap *map, Hashmap_traverse_cb traverse_cb)
{
    int i = 0;
    int j = 0;
    int rc = 0;

    for(i = 0; i < DArray_count(map->buckets); i++) {
        DArray *bucket = DArray_get(map->buckets, i);
        if(bucket) {
            for(j = 0; j < DArray_count(bucket); j++) {
                HashmapNode *node = DArray_get(bucket, j);
                rc = traverse_cb(node);
                if(rc != 0) return rc;
            }
        }
    }

    return 0;
}

void *Hashmap_delete(Hashmap *map, void *key)
{
    uint32_t hash = 0;
    DArray *bucket = Hashmap_find_bucket(map, key, 0, &hash);
    if(!bucket) return NULL;

    int i = Hashmap_get_node(map, hash, bucket, key);
    if(i == -1) return NULL;

    HashmapNode *node = DArray_get(bucket, i);
    void *data = node->data;
    free(node);

    HashmapNode *ending = DArray_pop(bucket);

    if(ending != node) {
        // alright looks like it's not the last one, swap it
        DArray_set(bucket, i, ending);
    }

    return data;
}
```

这个实现中并没有什么复杂的东西，但是 `default_hash` 和 `Hashmap_find_bucket` 需要一些解释。当你使用 `Hashmap_create` 时，你可以传入任何定制的比较和哈希函数。但是如果你没有则会使用 `default_compare` 和 `default_hash` 函数。

需要观察的第一件事，是`default_hash`的行为。这是一个简单的哈希函数，叫做“Jenkins hash”，以Bob Jenkins的名字命名。我从维基百科上获得了这个算法。它仅仅遍历键（`bstring`）的每个字节来计算哈希，以便得出`uint32_t`的结果。它使用一些加法和异或运算来实现。

哈希函数有很多中，它们具有不同的特性，然而一旦你选择了一种，就需要一种方法来使用它找到正确的桶。`Hashmap_find_bucket`像这样实现它：

- 首先调用`map->hash(key)`来获得键的哈希值。
- 之后使用`hash % DEFAULT_NUMBER_OF_BUCKETS`，这样无论哈希值有多大，都能找到匹配的桶。
- 找到桶之后，它是个`DArray`，可能还没有创建，这取决与`create`变量的内容。
- 一旦找到了正确的`DArray`桶，就会将它返回，并且`hash_out`变量用于向调用者提供所找到的哈希值。

其它函数都使用`Hashmap_find_bucket`来完成工作：

- 设置键值对涉及到找到正确的桶，之后创建`HashmapNode`，将它添加到桶中。
- 获取键值涉及到找到正确的桶，之后找到匹配`hash`和`key`的`HashmapNode`。
- 删除元素也需要找到正确的桶，找到所需的节点，之后通过与末尾的节点交换位置来删除。

你需要学习的唯一一个其他函数是`Hashmap_travers`，它仅仅遍历每个桶，对于任何含有值的桶，在每个值上调用`traverse_cb`。这就是扫描整个`Hashmap`的办法。

单元测试

最后你需要编写单元测试，对于所有这些操作：

```
#include "minunit.h"
#include <lcthw/hashmap.h>
#include <assert.h>
#include <lcthw/bstrlib.h>

Hashmap *map = NULL;
static int traverse_called = 0;
struct tagbstring test1 = bsStatic("test data 1");
struct tagbstring test2 = bsStatic("test data 2");
struct tagbstring test3 = bsStatic("xest data 3");
struct tagbstring expect1 = bsStatic("THE VALUE 1");
struct tagbstring expect2 = bsStatic("THE VALUE 2");
struct tagbstring expect3 = bsStatic("THE VALUE 3");

static int traverse_good_cb(HashmapNode *node)
{
```

```
    debug("KEY: %s", bdata((bstring)node->key));
    traverse_called++;
    return 0;
}


static int traverse_fail_cb(HashmapNode *node)
{
    debug("KEY: %s", bdata((bstring)node->key));
    traverse_called++;

    if(traverse_called == 2) {
        return 1;
    } else {
        return 0;
    }
}


char *test_create()
{
    map = Hashmap_create(NULL, NULL);
    mu_assert(map != NULL, "Failed to create map.");

    return NULL;
}

char *test_destroy()
{
    Hashmap_destroy(map);

    return NULL;
}


char *test_get_set()
{
    int rc = Hashmap_set(map, &test1, &expect1);
    mu_assert(rc == 0, "Failed to set &test1");
    bstring result = Hashmap_get(map, &test1);
    mu_assert(result == &expect1, "Wrong value for test1.");

    rc = Hashmap_set(map, &test2, &expect2);
    mu_assert(rc == 0, "Failed to set test2");
    result = Hashmap_get(map, &test2);
    mu_assert(result == &expect2, "Wrong value for test2.");

    rc = Hashmap_set(map, &test3, &expect3);
    mu_assert(rc == 0, "Failed to set test3");
    result = Hashmap_get(map, &test3);
    mu_assert(result == &expect3, "Wrong value for test3.");

    return NULL;
```

```
}

char *test_traverse()
{
    int rc = Hashmap_traverse(map, traverse_good_cb);
    mu_assert(rc == 0, "Failed to traverse.");
    mu_assert(traverse_called == 3, "Wrong count traverse.");

    traverse_called = 0;
    rc = Hashmap_traverse(map, traverse_fail_cb);
    mu_assert(rc == 1, "Failed to traverse.");
    mu_assert(traverse_called == 2, "Wrong count traverse for fa
il.");

    return NULL;
}

char *test_delete()
{
    bstring deleted = (bstring)Hashmap_delete(map, &test1);
    mu_assert(deleted != NULL, "Got NULL on delete.");
    mu_assert(deleted == &expect1, "Should get test1");
    bstring result = Hashmap_get(map, &test1);
    mu_assert(result == NULL, "Should delete.");

    deleted = (bstring)Hashmap_delete(map, &test2);
    mu_assert(deleted != NULL, "Got NULL on delete.");
    mu_assert(deleted == &expect2, "Should get test2");
    result = Hashmap_get(map, &test2);
    mu_assert(result == NULL, "Should delete.");

    deleted = (bstring)Hashmap_delete(map, &test3);
    mu_assert(deleted != NULL, "Got NULL on delete.");
    mu_assert(deleted == &expect3, "Should get test3");
    result = Hashmap_get(map, &test3);
    mu_assert(result == NULL, "Should delete.");

    return NULL;
}

char *all_tests()
{
    mu_suite_start();

    mu_run_test(test_create);
    mu_run_test(test_get_set);
    mu_run_test(test_traverse);
    mu_run_test(test_delete);
    mu_run_test(test_destroy);

    return NULL;
}
```

```
RUN_TESTS(all_tests);
```

需要学习的唯一一件事情就是我在单元测试的顶端使用了 `bstring` 的特性来创建静态字符串用于测试。我使用 `tagbstring` 和 `bsStatic` 在7~13行创建他们。

如何改进

这是一个非常简单的 `Hashmap` 实现，就像书中的大多数其他数据结构那样。我的目标不是让你以非常快的速度来掌握数据结构。通常这些讨论起来非常复杂，并且会让你偏离真正的基础和实用的数据结构。我的目标是提供一个易于理解的起始点，然后再改进或理解它们如何实现。

对于这和练习，下面是你能够用于改进这个实现的方法：

- 你可以对每个桶进行排序，使它们有序。这会增加你的插入时间，但是减少寻找时间，因为你可以使用二分搜索来寻找每个节点。到现在为止它遍历桶中的所有节点来寻找元素。
- 你可以动态设定桶的数量，或者让调用者指定每个 `Hashmap` 中的该值。
- 你可以使用更好的 `default_hash` 函数，有许多这样的函数。
- 这个实现以及几乎所有实现都有将一些特定的键存到一个桶中的风险。这会使你的程序运行速度变慢，因为它使 `Hashmap` 的处理过程变成了处理单个的 `DArray` 。如果你对桶中的数组排序会有帮助，但是你可以仅仅使用更好的哈希函数来避免，并且对于真正的偏执狂，你可以添加一个随机的盐，让键不可预测。
- 你可以删掉不歪有任何节点的桶来节约空间，或者将空的桶当如缓存中，便于节约创建和销毁它们的开销。
- 现在为止它可以添加已存在的元素，编写一个替代的实现，使它只能够添加不存在的元素。

像往常一样，你需要浏览每个函数，并且使之健壮。 `Hashmap` 也可以使用一些调试设置，来执行不变量检查。

附加题

- 研究你最喜欢的编程语言的 `Hashmap` 实现，了解它们具有什么特性。
- 找到 `Hashmap` 的主要缺点，以及如何避免它们。例如，它们不做特定的修改就不能保存顺序，并且当你基于键的一部分来查找元素时，它们就不能生效。
- 编写单元测试来展示将键都填充到 `Hashmap` 的一个桶中所带来的缺陷，之后测试这样如何影响性能。一个实现它的好方法，就是把桶的数量减少到一个愚蠢的数值，比如1。

练习38：哈希算法

> 原文：Exercise 38: Hashmap Algorithms
>
> 译者：飞龙

你需要在这个练习中实现下面这三个哈希函数：

FNV-1a

以创造者Glenn Fowler、Phong Vo 和 Landon Curt Noll的名字命名。这个算法产生合理的数值并且相当快。

Adler-32

以Mark Adler命名。一个比较糟糕的算法，但是由来已久并且适于学习。

DJB Hash

由Dan J. Bernstein (DJB)发明的哈希算法，但是难以找到这个算法的讨论。它非常快，但是结果不是很好。

你应该看到我使用了Jenkins hash作为 `Hashmap` 数据结构的默认哈希函数，所以这个练习的重点会放在这三个新的函数上。它们的代码通常来说不多，并且没有任何优化。像往常一样我会放慢速度来让你理解。

头文件非常简单，所以我以它开始：

```c
#ifndef hashmap_algos_h
#define hashmap_algos_h

#include <stdint.h>

uint32_t Hashmap_fnv1a_hash(void *data);

uint32_t Hashmap_adler32_hash(void *data);

uint32_t Hashmap_djb_hash(void *data);

#endif
```

我只是声明了三个函数，我会在 `hashmap_algos.c` 文件中实现它们：

```c
#include <lcthw/hashmap_algos.h>
#include <lcthw/bstrlib.h>

// settings taken from
// http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-param
```

```c
const uint32_t FNV_PRIME = 16777619;
const uint32_t FNV_OFFSET_BASIS = 2166136261;

uint32_t Hashmap_fnv1a_hash(void *data)
{
    bstring s = (bstring)data;
    uint32_t hash = FNV_OFFSET_BASIS;
    int i = 0;

    for(i = 0; i < blength(s); i++) {
        hash ^= bchare(s, i, 0);
        hash *= FNV_PRIME;
    }

    return hash;
}

const int MOD_ADLER = 65521;

uint32_t Hashmap_adler32_hash(void *data)
{
    bstring s = (bstring)data;
    uint32_t a = 1, b = 0;
    int i = 0;

    for (i = 0; i < blength(s); i++)
    {
        a = (a + bchare(s, i, 0)) % MOD_ADLER;
        b = (b + a) % MOD_ADLER;
    }

    return (b << 16) | a;
}

uint32_t Hashmap_djb_hash(void *data)
{
    bstring s = (bstring)data;
    uint32_t hash = 5381;
    int i = 0;

    for(i = 0; i < blength(s); i++) {
        hash = ((hash << 5) + hash) + bchare(s, i, 0); /* hash *
 33 + c */
    }

    return hash;
}
```

这个文件中有三个哈希函数。你应该注意到我默认使用 `bstring` 作为键，并且使用了 `bchare` 函数从字符串获取字符，然而如果字符超出了字符串的长度会返回0。

这些算法中每个都可以在网上搜索到，所以你需要搜索它们并阅读相关内容。同时我主要使用维基百科上的结果，之后参照了其它来源。

接着我为每个算法编写了单元测试，同时也测试了它们在多个桶中的分布情况。

```
#include <lcthw/bstrlib.h>
#include <lcthw/hashmap.h>
#include <lcthw/hashmap_algos.h>
#include <lcthw/darray.h>
#include "minunit.h"

struct tagbstring test1 = bsStatic("test data 1");
struct tagbstring test2 = bsStatic("test data 2");
struct tagbstring test3 = bsStatic("xest data 3");

char *test_fnv1a()
{
    uint32_t hash = Hashmap_fnv1a_hash(&test1);
    mu_assert(hash != 0, "Bad hash.");

    hash = Hashmap_fnv1a_hash(&test2);
    mu_assert(hash != 0, "Bad hash.");

    hash = Hashmap_fnv1a_hash(&test3);
    mu_assert(hash != 0, "Bad hash.");

    return NULL;
}

char *test_adler32()
{
    uint32_t hash = Hashmap_adler32_hash(&test1);
    mu_assert(hash != 0, "Bad hash.");

    hash = Hashmap_adler32_hash(&test2);
    mu_assert(hash != 0, "Bad hash.");

    hash = Hashmap_adler32_hash(&test3);
    mu_assert(hash != 0, "Bad hash.");

    return NULL;
}

char *test_djb()
{
    uint32_t hash = Hashmap_djb_hash(&test1);
    mu_assert(hash != 0, "Bad hash.");

    hash = Hashmap_djb_hash(&test2);
    mu_assert(hash != 0, "Bad hash.");

    hash = Hashmap_djb_hash(&test3);
```

```
    mu_assert(hash != 0, "Bad hash.");

    return NULL;
}

#define BUCKETS 100
#define BUFFER_LEN 20
#define NUM_KEYS BUCKETS * 1000
enum { ALGO_FNV1A, ALGO_ADLER32, ALGO_DJB};

int gen_keys(DArray *keys, int num_keys)
{
    int i = 0;
    FILE *urand = fopen("/dev/urandom", "r");
    check(urand != NULL, "Failed to open /dev/urandom");

    struct bStream *stream = bsopen((bNread)fread, urand);
    check(stream != NULL, "Failed to open /dev/urandom");

    bstring key = bfromcstr("");
    int rc = 0;

    // FNV1a histogram
    for(i = 0; i < num_keys; i++) {
        rc = bsread(key, stream, BUFFER_LEN);
        check(rc >= 0, "Failed to read from /dev/urandom.");

        DArray_push(keys, bstrcpy(key));
    }

    bsclose(stream);
    fclose(urand);
    return 0;

error:
    return -1;
}

void destroy_keys(DArray *keys)
{
    int i = 0;
    for(i = 0; i < NUM_KEYS; i++) {
        bdestroy(DArray_get(keys, i));
    }

    DArray_destroy(keys);
}

void fill_distribution(int *stats, DArray *keys, Hashmap_hash ha
sh_func)
{
    int i = 0;
    uint32_t hash = 0;
```

```c
    for(i = 0; i < DArray_count(keys); i++) {
        hash = hash_func(DArray_get(keys, i));
        stats[hash % BUCKETS] += 1;
    }

}

char *test_distribution()
{
    int i = 0;
    int stats[3][BUCKETS] = {{0}};
    DArray *keys = DArray_create(0, NUM_KEYS);

    mu_assert(gen_keys(keys, NUM_KEYS) == 0, "Failed to generate
 random keys.");

    fill_distribution(stats[ALGO_FNV1A], keys, Hashmap_fnv1a_has
h);
    fill_distribution(stats[ALGO_ADLER32], keys, Hashmap_adler32
_hash);
    fill_distribution(stats[ALGO_DJB], keys, Hashmap_djb_hash);

    fprintf(stderr, "FNV\tA32\tDJB\n");

    for(i = 0; i < BUCKETS; i++) {
        fprintf(stderr, "%d\t%d\t%d\n",
                stats[ALGO_FNV1A][i],
                stats[ALGO_ADLER32][i],
                stats[ALGO_DJB][i]);
    }

    destroy_keys(keys);

    return NULL;
}

char *all_tests()
{
    mu_suite_start();

    mu_run_test(test_fnv1a);
    mu_run_test(test_adler32);
    mu_run_test(test_djb);
    mu_run_test(test_distribution);

    return NULL;
}

RUN_TESTS(all_tests);
```

我在代码中将 `BUCKETS` 的值设置得非常高，因为我的电脑足够快。如果你将它和 `NUM_KEYS` 调低，就会比较慢了。这个测试运行之后，对于每个哈希函数，通过使用R语言做统计分析，可以观察键的分布情况。

我实现它的方式是使用 `gen_keys` 函数生成键的大型列表。这些键从 `/dev/urandom` 设备中获得，它们是一些随机的字节。之后我使用了这些键来调用 `fill_distribution` ，填充了 `stats` 数组，这些键计算哈希值后会被放入理论上的一些桶中。所有这类函数会遍历所有键，计算哈希，之后执行类似 `Hashmap` 所做的事情来寻找正确的桶。

最后我只是简单打印出一个三列的表格，包含每个桶的最终数量，展示了每个桶中随机储存了多少个键。之后可以观察这些数值，来判断这些哈希函数是否合理对键进行分配。

你会看到什么

教授R是这本书范围之外的内容，但是如果你想试试它，可以访问r-project.org。

下面是一个简略的shell会话，向你展示了我如何运行 `1tests/hashmap_algos_test` 来获取 `test_distribution` 产生的表（这里没有展示），之后使用R来观察统计结果：

```
$ tests/hashmap_algos_tests
# copy-paste the table it prints out
$ vim hash.txt
$ R
> hash <- read.table("hash.txt", header=T)
> summary(hash)
      FNV             A32              DJB
 Min.   : 945   Min.   : 908.0   Min.   : 927
 1st Qu.: 980   1st Qu.: 980.8   1st Qu.: 979
 Median : 998   Median :1000.0   Median : 998
 Mean   :1000   Mean   :1000.0   Mean   :1000
 3rd Qu.:1016   3rd Qu.:1019.2   3rd Qu.:1021
 Max.   :1072   Max.   :1075.0   Max.   :1082
```

首先我只是运行测试，它会在屏幕上打印表格。之后我将它复制粘贴到下来并使用 `vim hash.txt` 来储存数据。如果你观察数据，它会带有显示这三个算法的 `FNV A32 DJB` 表头。

接着，我运行R来使用 `read.table` 命令加载数据集。它是个非常智能的函数，适用于这种tab分隔的数据，我只要告诉它 `header=T` ，它就知道数据集中带有表头。

最后，我家在了数据并且可以使用 `summary` 来打印出它每行的统计结果。这里你可以看到每个函数处理随机数据实际上都没有问题。我会解释每个行的意义：

Min.

它是列出数据的最小值。FNV似乎在这方面是最优的，因为它有最大的结果，也就是说它的下界最严格。

1st Qu.

数据的第一个四分位点。

Median

如果你对它们排序，这个数值就是最重点的那个数。中位数比起均值来讲更有用一些。

Mean

均值对大多数人意味着“平均”，它是数据的总数比数量。如果你观察它们，所有均值都是1000，这非常棒。如果你将它去中位数对比，你会发现，这三个中位数都很接近均值。这就意味着这些数据都没有“偏向”一端，所以均值是可信的。

3rd Qu.

数据后四分之一的起始点，代表了尾部的数值。

Max.

这是数据中的最大值，代表了它们的上界。

观察这些数据，你会发现这些哈希算法似乎都适用于随机的键，并且均值与我设置的 `NUM_KEYS` 匹配。我所要找的就是如果我为每个桶中生成了1000个键，那么平均每个桶中就应该有100个键。如果哈希函数工作不正常，你会发现统计结果中均值不是1000，并且第一个和第三个四分位点非常高。一个好的哈希算法应该使平均值为1000，并且具有严格的范围。

同时，你应该明白即使在这个单元测试的不同运行之间，你的数据的大多数应该和我不同。

如何使它崩溃

这个练习的最后，我打算向你介绍使它崩溃的方法。我需要让你变写你能编写的最烂的哈希函数，并且我会使用数据来证明它确实很烂。你可以使用R来进行统计，就像我上面一样，但也可能你知道其他可以使用的工具来进行相同的统计操作。

这里的目标是让一个哈希函数，它表面看起来是正常的，但实际运行就得到一个糟糕的均值，并且分布广泛。这意味着你不能只让你返回1，而是需要返回一些看似正常的数值，但是分布广泛并且都填充到相同的桶中。

如果你对这四个函数之一做了一些小修改来完成任务，我会给你额外的分数。

这个练习的目的是，想像一下一些“友好”的程序员见到你并且打算改进你的哈希函数，但是实际上只是留了个把你的 `Hashmap` 搞砸的后门。

附加题

- 将`hashmap.c`中的`default_hash`换成`hashmap_algos.c`中的算法之一，并且再次通过所有测试。
- 向`hashmap_algos_tests.c`添加`default_hash`，并将它与其它三个哈希函数比较。
- 寻找一些更多的哈希函数并添加进来，你永远都不可能找到太多的哈希函数！

练习39：字符串算法

> 原文：Exercise 39: String Algorithms
>
> 译者：飞龙

这个练习中，我会向你展示可能是最快的字符串搜索算法之一，并且将它与 `bstrlib.c` 中现有的 `binstr` 比较。 `binstr` 的文档说它仅仅使用了“暴力搜索”的字符串算法来寻找第一个实例。我所实现的函数使用Boyer-Moore-Horspool（BMH）算法，如果你分析理论时间的话，一般认为它会更快。你也会看到，如果我的实现没有任何缺陷，BMH的实际时间会比 `binstr` 简单的暴力搜索更糟。

这个练习的要点并不是真正解释算法本身，因为你可以直接去Boyer-Moore-Horspool 的维基百科页面去阅读它。这个算法的要点就是它会计算出“跳跃字符列表”作为第一步操作，之后它使用这个列表来快速扫描整个字符串。它应当比暴力搜索更快，所以让我们在文件里写出代码来看看吧。

首先，创建头文件：

```
#ifndef string_algos_h
#define string_algos_h

#include <lcthw/bstrlib.h>
#include <lcthw/darray.h>

typedef struct StringScanner {
    bstring in;
    const unsigned char *haystack;
    ssize_t hlen;
    const unsigned char *needle;
    ssize_t nlen;
    size_t skip_chars[UCHAR_MAX + 1];
} StringScanner;

int String_find(bstring in, bstring what);

StringScanner *StringScanner_create(bstring in);

int StringScanner_scan(StringScanner *scan, bstring tofind);

void StringScanner_destroy(StringScanner *scan);

#endif
```

为了观察“跳跃字符列表”的效果，我打算创建这个算法的两种版本：

String_find

只是在一个字符串中，寻找另一个字符串的首个实例，以一个动作执行整个算法。

StringScanner_scan

使用 `StringScanner` 状态结构，将跳跃列表的构建和实际的查找操作分开。这让我能看到什么影响了性能。这个模型有另一个优点，就是我可以在一个字符串中逐步搜索，并且快速地找到所有实例。

一旦你完成了头文件，下面就是实现了：

```
#include <lcthw/string_algos.h>
#include <limits.h>

static inline void String_setup_skip_chars(
        size_t *skip_chars,
        const unsigned char *needle, ssize_t nlen)
{
    size_t i = 0;
    size_t last = nlen - 1;

    for(i = 0; i < UCHAR_MAX + 1; i++) {
        skip_chars[i] = nlen;
    }

    for (i = 0; i < last; i++) {
        skip_chars[needle[i]] = last - i;
    }
}


static inline const unsigned char *String_base_search(
        const unsigned char *haystack, ssize_t hlen,
        const unsigned char *needle, ssize_t nlen,
        size_t *skip_chars)
{
    size_t i = 0;
    size_t last = nlen - 1;

    assert(haystack != NULL && "Given bad haystack to search.");
    assert(needle != NULL && "Given bad needle to search for.");

    check(nlen > 0, "nlen can't be <= 0");
    check(hlen > 0, "hlen can't be <= 0");

    while (hlen >= nlen)
    {
        for (i = last; haystack[i] == needle[i]; i--) {
            if (i == 0) {
                return haystack;
            }
        }
```

```
        hlen -= skip_chars[haystack[last]];
        haystack += skip_chars[haystack[last]];
    }

error: // fallthrough
    return NULL;
}

int String_find(bstring in, bstring what)
{
    const unsigned char *found = NULL;

    const unsigned char *haystack = (const unsigned char *)bdata
(in);
    ssize_t hlen = blength(in);
    const unsigned char *needle = (const unsigned char *)bdata(w
hat);
    ssize_t nlen = blength(what);
    size_t skip_chars[UCHAR_MAX + 1] = {0};

    String_setup_skip_chars(skip_chars, needle, nlen);

    found = String_base_search(haystack, hlen, needle, nlen, ski
p_chars);

    return found != NULL ? found - haystack : -1;
}

StringScanner *StringScanner_create(bstring in)
{
    StringScanner *scan = calloc(1, sizeof(StringScanner));
    check_mem(scan);

    scan->in = in;
    scan->haystack = (const unsigned char *)bdata(in);
    scan->hlen = blength(in);

    assert(scan != NULL && "fuck");
    return scan;

error:
    free(scan);
    return NULL;
}

static inline void StringScanner_set_needle(StringScanner *scan,
 bstring tofind)
{
    scan->needle = (const unsigned char *)bdata(tofind);
    scan->nlen = blength(tofind);

    String_setup_skip_chars(scan->skip_chars, scan->needle, scan
->nlen);
```

```
}

static inline void StringScanner_reset(StringScanner *scan)
{
    scan->haystack = (const unsigned char *)bdata(scan->in);
    scan->hlen = blength(scan->in);
}

int StringScanner_scan(StringScanner *scan, bstring tofind)
{
    const unsigned char *found = NULL;
    ssize_t found_at = 0;

    if(scan->hlen <= 0) {
        StringScanner_reset(scan);
        return -1;
    }

    if((const unsigned char *)bdata(tofind) != scan->needle) {
        StringScanner_set_needle(scan, tofind);
    }

    found = String_base_search(
            scan->haystack, scan->hlen,
            scan->needle, scan->nlen,
            scan->skip_chars);

    if(found) {
        found_at = found - (const unsigned char *)bdata(scan->in
);
        scan->haystack = found + scan->nlen;
        scan->hlen -= found_at - scan->nlen;
    } else {
        // done, reset the setup
        StringScanner_reset(scan);
        found_at = -1;
    }

    return found_at;
}


void StringScanner_destroy(StringScanner *scan)
{
    if(scan) {
        free(scan);
    }
}
```

整个算法都在两个`static inline`的函数中，叫做`String_setup_skip_chars`和`String_base_search`。它们在别的函数中使用，用于实现我想要的的搜索形式。研究这两个函数，并且与维基百科的描述对比，你就可以知道它的工作原理。

之后 `String_find` 使用这两个函数来寻找并返回所发现的位置。它非常简单并且我使用它来查看“跳跃字符列表”的构建如何影响到真实性能。要注意，你或许可以使它更快，但是我要教给你在你实现算法之后如何验证理论速度。

`StringScanner_scan` 函数随后按照“创建、扫描、销毁”的常用模式，并且用于在一个字符串中逐步搜索另一个字符串。当我向你展示单元测试的时候，你会看到它如何使用。

最后，我编写了单元测试来确保算法有效，之后在它的注释部分，我为三个搜索函数运行了简单的性能测试：

```
#include "minunit.h"
#include <lcthw/string_algos.h>
#include <lcthw/bstrlib.h>
#include <time.h>

struct tagbstring IN_STR = bsStatic("I have ALPHA beta ALPHA and
 oranges ALPHA");
struct tagbstring ALPHA = bsStatic("ALPHA");
const int TEST_TIME = 1;

char *test_find_and_scan()
{
    StringScanner *scan = StringScanner_create(&IN_STR);
    mu_assert(scan != NULL, "Failed to make the scanner.");

    int find_i = String_find(&IN_STR, &ALPHA);
    mu_assert(find_i > 0, "Failed to find 'ALPHA' in test string
.");

    int scan_i = StringScanner_scan(scan, &ALPHA);
    mu_assert(scan_i > 0, "Failed to find 'ALPHA' with scan.");
    mu_assert(scan_i == find_i, "find and scan don't match");

    scan_i = StringScanner_scan(scan, &ALPHA);
    mu_assert(scan_i > find_i, "should find another ALPHA after
the first");

    scan_i = StringScanner_scan(scan, &ALPHA);
    mu_assert(scan_i > find_i, "should find another ALPHA after
the first");

    mu_assert(StringScanner_scan(scan, &ALPHA) == -1, "shouldn't
 find it");

    StringScanner_destroy(scan);

    return NULL;
}

char *test_binstr_performance()
{
```

```
    int i = 0;
    int found_at = 0;
    unsigned long find_count = 0;
    time_t elapsed = 0;
    time_t start = time(NULL);

    do {
        for(i = 0; i < 1000; i++) {
            found_at = binstr(&IN_STR, 0, &ALPHA);
            mu_assert(found_at != BSTR_ERR, "Failed to find!");
            find_count++;
        }

        elapsed = time(NULL) - start;
    } while(elapsed <= TEST_TIME);

    debug("BINSTR COUNT: %lu, END TIME: %d, OPS: %f",
            find_count, (int)elapsed, (double)find_count / elaps
ed);
    return NULL;
}

char *test_find_performance()
{
    int i = 0;
    int found_at = 0;
    unsigned long find_count = 0;
    time_t elapsed = 0;
    time_t start = time(NULL);

    do {
        for(i = 0; i < 1000; i++) {
            found_at = String_find(&IN_STR, &ALPHA);
            find_count++;
        }

        elapsed = time(NULL) - start;
    } while(elapsed <= TEST_TIME);

    debug("FIND COUNT: %lu, END TIME: %d, OPS: %f",
            find_count, (int)elapsed, (double)find_count / elaps
ed);

    return NULL;
}

char *test_scan_performance()
{
    int i = 0;
    int found_at = 0;
    unsigned long find_count = 0;
    time_t elapsed = 0;
    StringScanner *scan = StringScanner_create(&IN_STR);
```

```
    time_t start = time(NULL);

    do {
        for(i = 0; i < 1000; i++) {
            found_at = 0;

            do {
                found_at = StringScanner_scan(scan, &ALPHA);
                find_count++;
            } while(found_at != -1);
        }

        elapsed = time(NULL) - start;
    } while(elapsed <= TEST_TIME);

    debug("SCAN COUNT: %lu, END TIME: %d, OPS: %f",
            find_count, (int)elapsed, (double)find_count / elaps
ed);

    StringScanner_destroy(scan);

    return NULL;
}


char *all_tests()
{
    mu_suite_start();

    mu_run_test(test_find_and_scan);

    // this is an idiom for commenting out sections of code
#if 0
    mu_run_test(test_scan_performance);
    mu_run_test(test_find_performance);
    mu_run_test(test_binstr_performance);
#endif

    return NULL;
}

RUN_TESTS(all_tests);
```

我把它们写在 `#if 0` 中间，它是使用C预处理器来注释一段代码的方法。像这样输入，并且把它和 `#endif` 移除，你就可以运行性能测试。当你继续这本书时，需要简单地把它们再次注释，以防它们浪费你的开发时间。

这个单元测试没有什么神奇之处，它只是在尊换种调用每个不同的函数，循环需要持续足够长的时间来得到一个几秒的样本。第一个测试（ `test_find_and_scan` ）只是确保我所编写的代码正常工作，因为测试无效的

代码没有意义。之后，下面的三个函数使用三个函数中的每一个来执行大量的搜索。

需要注意的一个技巧是，我在 `start` 中存储了起始时间，之后一直循环到至少过了 `TEST_TIME` 秒。这确保了我能或得到足够好的样本用于比较三者。我之后会使用不同的 `TEST_TIME` 设置来运行测试，并且分析结果。

你会看到什么

当我在我的笔记本上运行测试时，我得到的数据是这样的：

```
$ ./tests/string_algos_tests
DEBUG tests/string_algos_tests.c:124: ----- RUNNING: ./tests/str
ing_algos_tests
----
RUNNING: ./tests/string_algos_tests
DEBUG tests/string_algos_tests.c:116:
----- test_find_and_scan
DEBUG tests/string_algos_tests.c:117:
----- test_scan_performance
DEBUG tests/string_algos_tests.c:105: SCAN COUNT: 110272000, END
 TIME: 2, OPS: 55136000.000000
DEBUG tests/string_algos_tests.c:118:
----- test_find_performance
DEBUG tests/string_algos_tests.c:76: FIND COUNT: 12710000, END T
IME: 2, OPS: 6355000.000000
DEBUG tests/string_algos_tests.c:119:
----- test_binstr_performance
DEBUG tests/string_algos_tests.c:54: BINSTR COUNT: 72736000, END
 TIME: 2, OPS: 36368000.000000
ALL TESTS PASSED
Tests run: 4
$
```

我看到了它，觉得每轮运行应该超过两秒。并且，我打算多次运行它，并且像之前一样使用R来验证。下面是我获得的10个样例，每个基本上是10秒：

```
scan find binstr
71195200 6353700 37110200
75098000 6358400 37420800
74910000 6351300 37263600
74859600 6586100 37133200
73345600 6365200 37549700
74754400 6358000 37162400
75343600 6630400 37075000
73804800 6439900 36858700
74995200 6384300 36811700
74781200 6449500 37383000
```

我在shell的一点点帮助下获取数据，之后编辑输出：

```
$ for i in 1 2 3 4 5 6 7 8 9 10; do echo "RUN --- $i" >> times.l
og; ./tests/string_algos_tests 2>&1 | grep COUNT >> times.log ;
done
$ less times.log
$ vim times.log
```

现在你可以看到 `scan` 系统要优于另外两个，但是我会在R中打开它并且验证结果：

```
> times <- read.table("times.log", header=T)
> summary(times)
      scan                 find              binstr
 Min.    :71195200   Min.    :6351300   Min.    :36811700
 1st Qu.:74042200   1st Qu.:6358100   1st Qu.:37083800
 Median :74820400   Median :6374750   Median :37147800
 Mean   :74308760   Mean   :6427680   Mean   :37176830
 3rd Qu.:74973900   3rd Qu.:6447100   3rd Qu.:37353150
 Max.   :75343600   Max.   :6630400   Max.   :37549700
>
```

为了理解我为什么要生成这份概要统计，我必须对你解释一些统计学概念。我在这些数字中寻找的东西能够简单地告诉我，“这三个函数（`scan`、`find`、`binstr`）实际上不同吗？”我知道每次我运行测试函数的时候，我都会得到有些不同的数值，并且那些数值始终处理一个固定的范围。你可以看到两个四分位数反映了这一点。

我首先会去看均值，并且我会观察每个样例的均值是否不同于其它的。我可以清楚地看到 `scan` 优于 `binstr` ，同时后者优于 `find` 。然而问题来了，如果我只使用均值，就可以出现每个样例的范围会重叠的可能性。

如果均值不同，但是两个四分位点重叠会怎么用？这种情况下我只能说有这种可能性，并且如果我再次运行测试，均值就可能不同了。很可能出现的范围上的重叠是，我的两个样例（以及两个函数）并非实际上不同。任何我看到的差异都是随机产生的结果。

统计学拥有大量工具来解决这一问题，但是在我们的例子中我可以仅仅观察两个四分位值，以及所有样例的均值。如果均值不同，并且四分位值不可能重叠，就可以说它们完全不同。

在我的三个样例中，我可以说 `scan` 、 `find` 和 `binstr` 都是不同的，范围上没有重叠，并且（最重要的是）我可以相信数据。

分析结果

从结果中可以看出 `String_find` 比其它两个更慢。实际上，我认为慢的原因是我实现的方式有些问题。然而当我将它与 `StringScanner_scan` 比较时，我发现正是构造跳跃列表的那一部分最消耗时间。并且它的功能比 `scan` 要少，因为它仅仅找到了第一个位置，而 `scan` 找到了全部。

我也可以发现 `scan` 以很大优势优于 `binstr` 。同时我可以说 `scan` 的功能比其他两个要多，速度也更快。

下面是这个分析的一些注解：

- 我可能将实现或测试弄乱了。现在我打算研究所有实现BMH的可能方式来改进它。我也会确保我所做的事情正确。
- 如果你修改了测试运行的时间，你会得到不同的结果。这就是我没有考虑的”热身“环节。
- `test_scan_performance` 单元测试和其它两个并不相同，但是它比其它测试做得更多（并且也是按照时间和操作数量计算的），所以他可能是合理的。
- 我只通过在一个字符串内搜索另一个来执行测试。我应该使所查找的字符串随机化，来移除它们的位置和长度，作为干扰因素。
- `binstr` 的实现可能比“暴力搜索”要好。（所以应该自己编写暴力搜索作为对照。）
- 我可能以不幸的顺序来执行这些函数，并且随机化首先运行的测试可能会得到更好的结果。

可以从中学到的是，你需要确保知己的性能，即使你“正确”实现了一个算法。在这里BMH算法应该优于 `binstr` 算法，但是一个简单的测试证明了它是错误。如果我没有这些测试，我可能就使用了一个劣等的算法实现而不自知。参照这些度量，我可以开始调优我的实现，或者只是抛弃它并寻找新的算法。

附加题

- 看看你能不能使 `Scan_find` 更快。为什么我的实现这么慢？
- 尝试一些不同的搜索时长，看看你是否能得到不同的数值。当你改变 `scan` 的测试时间时，时间的长度会有什么影响？对于这些结果你能得出什么结论？
- 修改单元测试，使它最开始执行每个函数一小段时间，来消除任何“热身”缓解。这样会修改所运行时长的依赖性吗？每秒可能出现多少次操作？
- 使单元测试中的所查找字符串随机化，之后测量你的得到的性能。一种实现它的方式就是使用 `bstrlib.h` 中的 `bsplit` 函数在空格处分割 `IN_STR` 。之后使用你得到的 `strList` 结构访问它返回的每个字符串。这也教给你如何使用 `bstrList` 操作进行字符串处理。
- 尝试一些不同顺序的测试，看看能否得到不同的结果。

练习40：二叉搜索树

> 原文：[Exercise 40: Binary Search Trees]
>
> 译者：飞龙

二叉树是最简单的树形数据结构，虽然它在许多语言中被哈希表取代，但仍旧对于一些应用很实用。二叉树的各种变体可用于一些非常实用东西，比如数据库的索引、搜索算法结构、以及图像处理。

我把我的二叉树叫做`BSTree`，描述它的最佳方法就是它是另一种`Hashmap`形式的键值对储存容器。它们的差异在于，哈希表为键计算哈希值来寻找位置，而二叉树将键与树中的节点进行对比，之后深入树中找到储存它的最佳位置，基于它与其它节点的关系。

在我真正解释它的工作原理之前，让我向你展示`bstree.h`头文件，便于你看到数据结构，之后我会用它来解释如何构建。

```c
#ifndef _lcthw_BSTree_h
#define _lcthw_BSTree_h


typedef int (*BSTree_compare)(void *a, void *b);

typedef struct BSTreeNode {
    void *key;
    void *data;

    struct BSTreeNode *left;
    struct BSTreeNode *right;
    struct BSTreeNode *parent;
} BSTreeNode;

typedef struct BSTree {
    int count;
    BSTree_compare compare;
    BSTreeNode *root;
} BSTree;

typedef int (*BSTree_traverse_cb)(BSTreeNode *node);

BSTree *BSTree_create(BSTree_compare compare);
void BSTree_destroy(BSTree *map);

int BSTree_set(BSTree *map, void *key, void *data);
void *BSTree_get(BSTree *map, void *key);

int BSTree_traverse(BSTree *map, BSTree_traverse_cb traverse_cb)
;

void *BSTree_delete(BSTree *map, void *key);

#endif
```

这遵循了我之前用过的相同模式，我创建了一个基容器叫做 `BSTree` ，它含有叫做 `BSTreeNode` 的节点，组成实际内容。厌倦了吗？是的，这种结构也没有什么高明之处。

最重要的部分是， `BSTreeNode` 如何配置，以及它如何用于进行每个操作：设置、获取和删除。我会首先讲解 `get` ，因为它是最简单的操作，并且我会在数据结构上手动操作：

- 我获得你要找的键，并且用根节点开始遍历，首先我将你的键与这个节点的键进行对比。
- 如果你的键小于 `node.key` ，我使用 `left` 指针来详细遍历。
- 如果你的键大于 `node.key` ，我使用 `right` 指针来详细遍历。
- 重复第二步和第三部，知道我找到了匹配 `node.key` 的节点，或者我遍历到了没有左子树或右子树的节点。这种情况我会返回 `node.data` ，其它情况会返

回 `NULL` 。

这就是 `get` 的全部操作，现在是 `set` ，它几乎执行相同的操作，除了你在寻找防止新节点的位置。

- 如果 `BSTree.root` 为空，就算是执行完成了。它就是第一个节点。
- 之后我会将你的键与 `node.key` 进行比对，从根节点开始。
- 如果你的键小于或等于 `node.key` ，我会遍历左子树，否则是右子树。
- 重复第三步，直到我到达了没有左子树或右子树的节点，但是这是我需要选择的方向。
- 我选择了这个方向（左或者右）来放置新的节点，并且将这个新节点的父节点设为我来时的上一个节点。当我删除它时，我会使用它的父节点。

这也解释了它如何工作。如果寻找一个节点涉及到按照键的对比来遍历左子树或右子树，那么设置一个节点涉及到相同的事情，直到我找到了一个位置，可以在其左子树或右子树上放置新的节点。

花一些时间在纸上画出一些树并且遍历一些节点来进行查找或设置，你就可以理解它如何工作。之后你要准备好来看一看实现，我在其中解释了删除操作。删除一个节点非常麻烦，因此它最适合逐行的代码分解。

```
#include <lcthw/dbg.h>
#include <lcthw/bstree.h>
#include <stdlib.h>
#include <lcthw/bstrlib.h>

static int default_compare(void *a, void *b)
{
    return bstrcmp((bstring)a, (bstring)b);
}


BSTree *BSTree_create(BSTree_compare compare)
{
    BSTree *map = calloc(1, sizeof(BSTree));
    check_mem(map);

    map->compare = compare == NULL ? default_compare : compare;

    return map;

error:
    if(map) {
        BSTree_destroy(map);
    }
    return NULL;
}

static int BSTree_destroy_cb(BSTreeNode *node)
{
    free(node);
```

```c
    return 0;
}

void BSTree_destroy(BSTree *map)
{
    if(map) {
        BSTree_traverse(map, BSTree_destroy_cb);
        free(map);
    }
}


static inline BSTreeNode *BSTreeNode_create(BSTreeNode *parent,
void *key, void *data)
{
    BSTreeNode *node = calloc(1, sizeof(BSTreeNode));
    check_mem(node);

    node->key = key;
    node->data = data;
    node->parent = parent;
    return node;

error:
    return NULL;
}


static inline void BSTree_setnode(BSTree *map, BSTreeNode *node,
void *key, void *data)
{
    int cmp = map->compare(node->key, key);

    if(cmp <= 0) {
        if(node->left) {
            BSTree_setnode(map, node->left, key, data);
        } else {
            node->left = BSTreeNode_create(node, key, data);
        }
    } else {
        if(node->right) {
            BSTree_setnode(map, node->right, key, data);
        } else {
            node->right = BSTreeNode_create(node, key, data);
        }
    }
}


int BSTree_set(BSTree *map, void *key, void *data)
{
    if(map->root == NULL) {
        // first so just make it and get out
```

```
        map->root = BSTreeNode_create(NULL, key, data);
        check_mem(map->root);
    } else {
        BSTree_setnode(map, map->root, key, data);
    }

    return 0;
error:
    return -1;
}

static inline BSTreeNode *BSTree_getnode(BSTree *map, BSTreeNode
 *node, void *key)
{
    int cmp = map->compare(node->key, key);

    if(cmp == 0) {
        return node;
    } else if(cmp < 0) {
        if(node->left) {
            return BSTree_getnode(map, node->left, key);
        } else {
            return NULL;
        }
    } else {
        if(node->right) {
            return BSTree_getnode(map, node->right, key);
        } else {
            return NULL;
        }
    }
}

void *BSTree_get(BSTree *map, void *key)
{
    if(map->root == NULL) {
        return NULL;
    } else {
        BSTreeNode *node = BSTree_getnode(map, map->root, key);
        return node == NULL ? NULL : node->data;
    }
}


static inline int BSTree_traverse_nodes(BSTreeNode *node, BSTree
_traverse_cb traverse_cb)
{
    int rc = 0;

    if(node->left) {
        rc = BSTree_traverse_nodes(node->left, traverse_cb);
        if(rc != 0) return rc;
    }
```

```c
    if(node->right) {
        rc = BSTree_traverse_nodes(node->right, traverse_cb);
        if(rc != 0) return rc;
    }

    return traverse_cb(node);
}

int BSTree_traverse(BSTree *map, BSTree_traverse_cb traverse_cb)
{
    if(map->root) {
        return BSTree_traverse_nodes(map->root, traverse_cb);
    }

    return 0;
}

static inline BSTreeNode *BSTree_find_min(BSTreeNode *node)
{
    while(node->left) {
        node = node->left;
    }

    return node;
}

static inline void BSTree_replace_node_in_parent(BSTree *map, BS
TreeNode *node, BSTreeNode *new_value)
{
    if(node->parent) {
        if(node == node->parent->left) {
            node->parent->left = new_value;
        } else {
            node->parent->right = new_value;
        }
    } else {
        // this is the root so gotta change it
        map->root = new_value;
    }

    if(new_value) {
        new_value->parent = node->parent;
    }
}

static inline void BSTree_swap(BSTreeNode *a, BSTreeNode *b)
{
    void *temp = NULL;
    temp = b->key; b->key = a->key; a->key = temp;
    temp = b->data; b->data = a->data; a->data = temp;
}
```

```c
static inline BSTreeNode *BSTree_node_delete(BSTree *map, BSTree
Node *node, void *key)
{
    int cmp = map->compare(node->key, key);

    if(cmp < 0) {
        if(node->left) {
            return BSTree_node_delete(map, node->left, key);
        } else {
            // not found
            return NULL;
        }
    } else if(cmp > 0) {
        if(node->right) {
            return BSTree_node_delete(map, node->right, key);
        } else {
            // not found
            return NULL;
        }
    } else {
        if(node->left && node->right) {
            // swap this node for the smallest node that is bigg
er than us
            BSTreeNode *successor = BSTree_find_min(node->right)
;
            BSTree_swap(successor, node);

            // this leaves the old successor with possibly a rig
ht child
            // so replace it with that right child
            BSTree_replace_node_in_parent(map, successor, succes
sor->right);

            // finally it's swapped, so return successor instead
 of node
            return successor;
        } else if(node->left) {
            BSTree_replace_node_in_parent(map, node, node->left)
;
        } else if(node->right) {
            BSTree_replace_node_in_parent(map, node, node->right
);
        } else {
            BSTree_replace_node_in_parent(map, node, NULL);
        }

        return node;
    }
}

void *BSTree_delete(BSTree *map, void *key)
{
    void *data = NULL;
```

```
    if(map->root) {
        BSTreeNode *node = BSTree_node_delete(map, map->root, ke
y);

        if(node) {
            data = node->data;
            free(node);
        }
    }

    return data;
}
```

在讲解 `BSTree_delete` 如何工作之前，我打算解释一下我用于执行递归函数的模式。你会发现许多树形数据结构都易于使用递归来编写，而写成单个函数的形式相当困难。一部分原因在于你需要为第一次操作建立一些初始的数据，之后在数据结构中递归，这难以写成一个函数。

解决办法就是使用两个函数。一个函数“建立”数据结构和首次递归的条件使第二层函数能够执行真正的逻辑。首先看一看 `BSTree_get` 来理解我所说的。

- 我设置了初始条件来处理递归，如果 `map->NULL` 是 `NULL` ，那么就返回 `NULL` 并且不需要递归。
- 之后我执行了真正的递归调用，它就是 `BSTree_getnode` 。我设置了根节点的初始条件、 `key` 和 `map` 。
- 之后在 `BSTree_getnode` 中，我执行了真正的递归逻辑，我将是用 `map->compare(node->key, key)` 来进行键的比对，并且根据结果遍历左子树或右子树，或者相等。
- 由于这个函数时“自相似”的，并且不用处理任何初始条件（因为 `BSTree_get` 处理了），我就可以使它非常简单。当它完成时会返回给调用者，最后把结构返回给 `BSTree_get` 。
- 最后，在结果不为 `NULL` 的情况下， `BSTree_get` 处理获得的 `node.data` 元素。

这种构造递归算法的方法，与我构造递归数据结构的方法一致。我创建了一个起始的“基函数”，它处理初始条件和一些边界情况，之后它调用了一个简洁的递归函数来执行任务。与之相比，我在 `BStree` 中创建了“基结构”，它持有递归的 `BSTreeNode` 结构，每个节点都引用树中的其它节点。使用这种模式让我更容易处理递归并保持简洁。

接下来，浏览 `BSTree_set` 和 `BSTree_setnode` ，来观察相同的模式。我使用 `BSTree_set` 来确保初始条件和便捷情况。常见的边界情况就是树中没有根节点，于是我需要创建一个函数来初始化它们。

这个模式适用于几乎任何递归的算法。我按照这种模式来编写它们：

- 理解初始变量，它们如何改变，以及递归每一步的终止条件。
- 编写调用自身的递归函数，带有参数作为终止条件和初始变量。

- 编程一个启动函数来设置算法的初始条件，并且处理边界情况，之后调用递归函数。
- 最后，启动函数返回最后的结果，并且如果递归函数不能处理最终的边界情况可能还要做调整。

这引导了我完成`BSTree_delete`和`BSTree_node_delete`。首先你可以看一下`BSTree_delete`和它的启动函数，它获取结果节点的数据，并且释放找到的节点。在`BSTree_node_delete`中事情就变得复杂了，因为要在树中任意位置删除一个节点，我需要将子节点翻转上来。我会逐行拆分这个函数：

bstree.c:190

我执行比较函数来找出应该选择的方向。

bstree.c:192-198

这是“小于”的分支，我应该移到左子树。这里左子树并不存在并且返回了`NULL`来表示“未找到”。这处理了一些不在`BSTree`中元素的删除操作。

bstree.c:199-205

和上面相同，但是是对于树的右侧分支。这就像其它函数一样只是在树中向下遍历，并且在不存在时返回`NULL`。

bstree.c:206

这里是发现目标节点的地方，因为键是相等的（`compare`返回了0）。

bstree.c:207

这个节点同时具有`left`和`right`分支，所以它深深嵌入在树中。

bstree.c:209

要移除这个节点，我首先要找到大于这个节点的最小节点，这里我在右子树上调用了`BSTree_find_min`。

bstree.c:210

一旦我获得了这个几点，我将它的`key`和`data`与当前节点互换。这样就高效地将当前节点移动到树的最底端，并且不同通过它的指针来调整节点。

bstree.c:214

现在`successor`是一个无效的分支，储存了当前节点的值。然而它可能还带有右子树，也就是说我必须做一个旋转使它的右节点上来代替它。

bstree.c:217

到此为止，`successor`已经从树中移出了，它的值被当前节点的值代替，它的任何子树都合并进了它的父节点。我可以像`node`一样返回它。

bstree.c:218

这个分支中，我了解到这个节点没有右子树只有左子树，所以我可以简单地用左节点来替代它。

bstree.c:219

我再次使用 `BSTree_replace_node_in_parent` 来执行替换，把左节点旋转上去。

bstree.c:220

这是只有右子树而没有左子树的情况，所以需要将右节点旋转上去。

bstree.c:221

再次使用相同的函数，这次是针对右节点。

bstree.c:222

最后，对于我发现的节点只剩下一种情况，就是它没有任何子树（没有做子树也没有右子树）。这种情况，我只需要使用相同函数以 `NULL` 来执行替换。

bstree.c:210

在此之后，我已经将当前节点从书中移除，并且以某个合适的子节点的元素来替换。我只需要把它返回给调用者，使它能够被释放或管理。

这个操作非常复杂，实话说，在一些树形数据结构中，我并不需要执行删除，而是把它当做软件中的常亮数据。如果我需要做繁杂的插入和删除工作，我会使用 `Hashmap` 。

最后，你可以查看它的单元测试以及测试方法：

```
#include "minunit.h"
#include <lcthw/bstree.h>
#include <assert.h>
#include <lcthw/bstrlib.h>
#include <stdlib.h>
#include <time.h>

BSTree *map = NULL;
static int traverse_called = 0;
struct tagbstring test1 = bsStatic("test data 1");
struct tagbstring test2 = bsStatic("test data 2");
struct tagbstring test3 = bsStatic("xest data 3");
struct tagbstring expect1 = bsStatic("THE VALUE 1");
struct tagbstring expect2 = bsStatic("THE VALUE 2");
struct tagbstring expect3 = bsStatic("THE VALUE 3");

static int traverse_good_cb(BSTreeNode *node)
{
    debug("KEY: %s", bdata((bstring)node->key));
    traverse_called++;
    return 0;
```

```
}


static int traverse_fail_cb(BSTreeNode *node)
{
    debug("KEY: %s", bdata((bstring)node->key));
    traverse_called++;

    if(traverse_called == 2) {
        return 1;
    } else {
        return 0;
    }
}


char *test_create()
{
    map = BSTree_create(NULL);
    mu_assert(map != NULL, "Failed to create map.");

    return NULL;
}

char *test_destroy()
{
    BSTree_destroy(map);

    return NULL;
}


char *test_get_set()
{
    int rc = BSTree_set(map, &test1, &expect1);
    mu_assert(rc == 0, "Failed to set &test1");
    bstring result = BSTree_get(map, &test1);
    mu_assert(result == &expect1, "Wrong value for test1.");

    rc = BSTree_set(map, &test2, &expect2);
    mu_assert(rc == 0, "Failed to set test2");
    result = BSTree_get(map, &test2);
    mu_assert(result == &expect2, "Wrong value for test2.");

    rc = BSTree_set(map, &test3, &expect3);
    mu_assert(rc == 0, "Failed to set test3");
    result = BSTree_get(map, &test3);
    mu_assert(result == &expect3, "Wrong value for test3.");

    return NULL;
}

char *test_traverse()
```

```
{
    int rc = BSTree_traverse(map, traverse_good_cb);
    mu_assert(rc == 0, "Failed to traverse.");
    mu_assert(traverse_called == 3, "Wrong count traverse.");

    traverse_called = 0;
    rc = BSTree_traverse(map, traverse_fail_cb);
    mu_assert(rc == 1, "Failed to traverse.");
    mu_assert(traverse_called == 2, "Wrong count traverse for fa
il.");

    return NULL;
}

char *test_delete()
{
    bstring deleted = (bstring)BSTree_delete(map, &test1);
    mu_assert(deleted != NULL, "Got NULL on delete.");
    mu_assert(deleted == &expect1, "Should get test1");
    bstring result = BSTree_get(map, &test1);
    mu_assert(result == NULL, "Should delete.");

    deleted = (bstring)BSTree_delete(map, &test1);
    mu_assert(deleted == NULL, "Should get NULL on delete");

    deleted = (bstring)BSTree_delete(map, &test2);
    mu_assert(deleted != NULL, "Got NULL on delete.");
    mu_assert(deleted == &expect2, "Should get test2");
    result = BSTree_get(map, &test2);
    mu_assert(result == NULL, "Should delete.");

    deleted = (bstring)BSTree_delete(map, &test3);
    mu_assert(deleted != NULL, "Got NULL on delete.");
    mu_assert(deleted == &expect3, "Should get test3");
    result = BSTree_get(map, &test3);
    mu_assert(result == NULL, "Should delete.");

    // test deleting non-existent stuff
    deleted = (bstring)BSTree_delete(map, &test3);
    mu_assert(deleted == NULL, "Should get NULL");

    return NULL;
}

char *test_fuzzing()
{
    BSTree *store = BSTree_create(NULL);
    int i = 0;
    int j = 0;
    bstring numbers[100] = {NULL};
    bstring data[100] = {NULL};
    srand((unsigned int)time(NULL));
```

```
    for(i = 0; i < 100; i++) {
        int num = rand();
        numbers[i] = bformat("%d", num);
        data[i] = bformat("data %d", num);
        BSTree_set(store, numbers[i], data[i]);
    }

    for(i = 0; i < 100; i++) {
        bstring value = BSTree_delete(store, numbers[i]);
        mu_assert(value == data[i], "Failed to delete the right
number.");

        mu_assert(BSTree_delete(store, numbers[i]) == NULL, "Sho
uld get nothing.");

        for(j = i+1; j < 99 - i; j++) {
            bstring value = BSTree_get(store, numbers[j]);
            mu_assert(value == data[j], "Failed to get the right
 number.");
        }

        bdestroy(value);
        bdestroy(numbers[i]);
    }

    BSTree_destroy(store);

    return NULL;
}

char *all_tests()
{
    mu_suite_start();

    mu_run_test(test_create);
    mu_run_test(test_get_set);
    mu_run_test(test_traverse);
    mu_run_test(test_delete);
    mu_run_test(test_destroy);
    mu_run_test(test_fuzzing);

    return NULL;
}

RUN_TESTS(all_tests);
```

我要重点讲解 `test_fuzzing` 函数，它是针对复杂数据结构的一种有趣的测试技巧。创建一些键来覆盖 `BSTree_node_delete` 的所有分支相当困难，而且有可能我会错过一些边界情况。更好的方法就是创建一个“模糊测试”的函数来执行所有操作，并尽可能以一种可怕且随机的方式执行它们。这里我插入了一系列随机字符串的键，之后我删除了它们并试着在删除之后获取它们的值。

这种测试可以避免只测试到你知道能正常工作的部分，这意味着你不会遗漏不知道的事情。通过想你的数据结构插入一些随机的垃圾数据，你可以碰到意料之外的事情，并检测出任何bug。

如何改进

不要完成下列任何习题，因为在下个练习中我会使用这里的单元测试，来教你使用一些性能调优的技巧。在你完成练习41之后，你需要返回来完成这些习题。

- 像之前一样，你应该执行所有防御性编程检查，并且为不应发生的情况添加`assert`。例如，你不应该在递归函数中获取到`NULL`，为此添加断言。
- 遍历函数按照左子树、右子树和当前节点的顺组进行遍历。你可以创建相反顺序的遍历函数。
- 每个节点上都会执行完整的字符串比较，但是我可以使用`Hashmap`的哈希函数来提升速度。我可以计算键的哈希值，在`BSTreeNode`中储存它。之后在每个创建的函数中，我可以实现计算出键的哈希值，然后在递归中向下传递。我可以使用哈希来很快地比较每个节点，就像`Hashmap`那样。

附加题

同样，现在先不要完成它们，直到完成练习41，那时你就可以使用`Valgrind`的性能调优技巧来完成它们了。

- 有一种不使用递归的替代的方法，也可以操作这个数据结构。维基百科上介绍了不使用递归来完成相同事情的替代方法。这样做会更好还是更糟？
- 查询你能找到的所有不同的树的相关资料。比如AVL树、红黑树、以及一些非树形结构例如跳转表。

练习41：将 Cachegrind 和 Callgrind 用于性能调优

> 原文：Exercise 41: Using Cachegrind And Callgrind For Performance Tuning
>
> 译者：飞龙

这个练习中，我打算上一节速成课，内容是使用 `Valgrind` 的两个工具 `callgrind` 和 `cachegrind` 。这两个工具会分析你程序的执行，并且告诉你哪一部分运行缓慢。这些结果非常精确，因为 `Valgrind` 的工作方式有助于你解决一些问题，比如执行过多的代码行，热点，内容访问问题，甚至是CPU的缓存未命中。

为了做这个练习，我打算使用 `bstree_tests` 单元测试，你之前用于寻找能提升算法的地方。你需要确保你这些程序的版本没有任何 `valgrind` 错误，并且和我的代码非常相似，因为我会使用我的代码的转储来谈论 `cachegrind` 和 `callgrind` 如何工作。

运行 Callgrind

为了运行Callgrind，你需要向 `valgrind` 传入 `--tool=callgrind` 选项，之后它会产生 `callgrind.out.PID` 文件（其中PID为所运行程序的进程PID）。一旦你这样运行了，你就可以使用一个叫做 `callgrind_annotate` 的工具分析 `callgrind.out` 文件，它会告诉你哪个函数运行中使用了最多的指令。下面是个例子，我在 `bstree_tests` 上运行了 `callgrind` ，之后得到了这个信息：

```
$ valgrind --dsymutil=yes --tool=callgrind tests/bstree_tests
...
$ callgrind_annotate callgrind.out.1232
----------------------------------------------------------------
----------------
Profile data file 'callgrind.out.1232' (creator: callgrind-3.7.0
.SVN)
----------------------------------------------------------------
----------------
I1 cache:
D1 cache:
LL cache:
Timerange: Basic block 0 - 1098689
Trigger: Program termination
Profiled target:  tests/bstree_tests (PID 1232, part 1)
Events recorded:  Ir
Events shown:     Ir
Event sort order: Ir
Thresholds:       99
Include dirs:
```

```
User annotated:
Auto-annotation:  off

--------------------------------------------------------------------------------
       Ir
--------------------------------------------------------------------------------
4,605,808  PROGRAM TOTALS

--------------------------------------------------------------------------------
       Ir  file:function
--------------------------------------------------------------------------------
  670,486  src/lcthw/bstrlib.c:bstrcmp [tests/bstree_tests]
  194,377  src/lcthw/bstree.c:BSTree_get [tests/bstree_tests]
   65,580  src/lcthw/bstree.c:default_compare [tests/bstree_tests]
   16,338  src/lcthw/bstree.c:BSTree_delete [tests/bstree_tests]
   13,000  src/lcthw/bstrlib.c:bformat [tests/bstree_tests]
   11,000  src/lcthw/bstrlib.c:bfromcstralloc [tests/bstree_tests]
    7,774  src/lcthw/bstree.c:BSTree_set [tests/bstree_tests]
    5,800  src/lcthw/bstrlib.c:bdestroy [tests/bstree_tests]
    2,323  src/lcthw/bstree.c:BSTreeNode_create [tests/bstree_tests]
    1,183  /private/tmp/pkg-build/coregrind//vg_preloaded.c:vg_cleanup_env [/usr/local/lib/valgrind/vgpreload_core-amd64-darwin.so]

$
```

我已经移除了单元测试和 `valgrind` 输出，因为它们对这个练习没有用。你应该看到了 `callgrind_anotate` 输出，它向你展示了每个函数所运行的指令数量（ `valgrind` 中叫做 `Ir` ），由高到低排序。你通常可以忽略头文件的数据，直接跳到函数列表。

> 注
>
> 如果你获取到一堆“???:Image”的行，并且它们不是你程序中的东西，那么你读到的是OS的垃圾。只需要在末尾添加 `| grep -v "???"` 来过滤掉它们。

我现在可以对这个输出做个简短的分解，来找出下一步观察什么：

- 每一行都列出了 `Ir` 序号和执行它们的 `file:function` 。 `Ir` 是指令数量，并且如果它越少就越快。这里有些复杂，但是首先要着眼于 `Ir` 。
- 解决这个程序的方式是观察最上面的函数，之后看看你首先可以改进哪一个。这里，我可以改进 `bstrcmp` 或者 `BStree_get` 。可能以 `BStree_get` 开始更容易些。
- 这些函数的一部分由单元测试调用，所以我可以忽略它们。类

似 `bformat` ， `bfromcstralloc` 和 `bdestroy` 就是这样的函数。
- 我也可以找到我可以简单地避免调用的函数。例如，或许我可以假设 `BSTree` 仅仅处理 `bstring` 键，之后我可以不使用回调系统，并且完全移除 `default_compare` 。

到目前为止，我只知道我打算改进 `BSTree_get` ，并且不是因为 `BSTree_get` 执行慢。这是分析的第二阶段。

Callgrind 注解源文件

下一步我使用 `callgrind_annotate` 输出 `bstree.c` 文件，并且使用所带有的 `Ir` 对每一行做注解。你可以通过运行下面的命令来得到注解后的源文件：

```
$ callgrind_annotate callgrind.out.1232 src/lcthw/bstree.c
...
```

你的输出会是这个源文件的一个较大的转储，但是我会将它们剪切成包含 `BSTree_get` 和 `BSTree_getnode` 的部分：

```
--------------------------------------------------------------------------------
-- User-annotated source: src/lcthw/bstree.c
--------------------------------------------------------------------------------
    Ir


 2,453  static inline BSTreeNode *BSTree_getnode(BSTree *map, BSTreeNode *node, void *key)
     .  {
61,853      int cmp = map->compare(node->key, key);
663,908  => src/lcthw/bstree.c:default_compare (14850x)
     .
14,850      if(cmp == 0) {
     .          return node;
24,794      } else if(cmp < 0) {
30,623          if(node->left) {
     .              return BSTree_getnode(map, node->left, key);
     .          } else {
     .              return NULL;
     .          }
     .      } else {
13,146          if(node->right) {
     .              return BSTree_getnode(map, node->right, key);
     .          } else {
     .              return NULL;
     .          }
     .      }
     .  }
     .
     .  void *BSTree_get(BSTree *map, void *key)
 4,912  {
24,557      if(map->root == NULL) {
14,736          return NULL;
     .      } else {
     .          BSTreeNode *node = BSTree_getnode(map, map->root, key);
 2,453          return node == NULL ? NULL : node->data;
     .      }
     .  }
```

每一行都显示它的`Ir`（指令）数量，或者一个点（`.`）来表示它并不重要。我所要找的就是一些热点，或者带有巨大数值的`Ir`的行，它能够被优化掉。这里，第十行的输出表明，`BSTree_getnode`开销非常大的原因是它调用了`default_comapre`，它又调用了`bstrcmp`。我已经知道了`bstrcmp`是性能最差的函数，所以如果我想要改进`BSTree_getnode`的速度，我应该首先解决掉它。

之后我以相同方式查看 `bstrcmp` ：

```
 98,370  int bstrcmp (const_bstring b0, const_bstring b1) {
      .  int i, v, n;
      .
196,740     if (b0 == NULL || b1 == NULL || b0->data == NULL ||
b1->data == NULL ||
 32,790             b0->slen < 0 || b1->slen < 0) return SHRT_MI
N;
 65,580     n = b0->slen; if (n > b1->slen) n = b1->slen;
 89,449     if (b0->slen == b1->slen && (b0->data == b1->data ||
 b0->slen == 0))
      .             return BSTR_OK;
      .
 23,915     for (i = 0; i < n; i ++) {
163,642             v = ((char) b0->data[i]) - ((char) b1->data[
i]);
      .             if (v != 0) return v;
      .             if (b0->data[i] == (unsigned char) '\0') ret
urn BSTR_OK;
      .     }
      .
      .     if (b0->slen > n) return 1;
      .     if (b1->slen > n) return -1;
      .     return BSTR_OK;
      .  }
```

输出中让我预料之外的事情就是 `bstrcmp` 最糟糕的一行并不是我想象中的字符比较。对于内存访问，顶部的防御性 `if` 语句将所有可能的无效变量都检查了一遍。与第十七行比较字符的语句相比，这个 `if` 语句进行了多于两倍的内存访问。如果我要优化 `bstcmp` ，我会完全把它去掉，或者在其它一些地方来执行它。

另一种选择是将这个检查改为 `assert` ，它只在开发时的运行中存在，之后在发布时把它去掉。我没有足够的证明来表明这行代码不适于这个数据结构，所以我可以证明移除它是可行的。

然而，我并不想弱化这个函数的防御性，来得到一些性能。在真实的性能优化环境，我会简单地把它放到列表中，之后挖掘程序中能得到的其它收益。

调优之道

> 我们应该忽略微小的效率，对于97%的情况：过早优化是万恶之源。
>
> -- 高德纳

在我看来，这个引述似乎忽略了一个关于性能调优的重点。在高德纳的话中，当你做性能调优时，如果你过早去做它，可能会导致各种问题。根据他的话，优化应该执行于“稍晚的某个时间”，或者这只是我的猜测。谁知道呢。

我打算澄清这个引述并不是完全错误，而是忽略了某些东西，并且我打算给出我的引述。你可以引用我的这段话：

> 使用证据来寻找最大的优化并花费最少的精力。
>
> -- 泽德 A. 肖

你什么时候优化并不重要，但是你需要弄清楚你的优化是否真正能改进软件，以及需要投入多少精力来实现它。通过证据你就可以找到代码中的位置，用一点点精力就能取得最大的提升。通常这些地方都是一些愚蠢的决定，就像 `bstrcmp` 试图检查任何东西不为 `NULL` 一样。

在某个特定时间点上，代码中需要调优的地方只剩下极其微小的优化，比如重新组织 `if` 语句，或者类似达夫设备这样的特殊循环。这时候，你应该停止优化，因为这是一个好机会，你可以通过重新设计软件并且避免这些事情来获得更多收益。

这是一些只想做优化的程序员没有看到的事情。许多时候，把一件事情做快的最好方法就是寻找避免它们的办法。在上面的分析中，我不打算优化 `bstrcmp` ，我会寻找一个不使用它的方法。也许我可以使用一种哈希算法来执行可排序的哈希计算而不是始终使用 `bstrcmp` 。也许我可以通过首先尝试第一个字符，如果它们不匹配就没必要调用 `bstrcmp` 。

如果在此之后你根本不能重新设计，那么就开始寻找微小的优化，但是要始终确保它们能够提升速度。要记住目标是使用最少的精力尽可能得到最大的效果。

使用 KCachegrind

这个练习最后一部分就是向你介绍一个叫做KCachegrind的神奇的GUI工具，用于分析 `callgrind` 和 `cachegrind` 的输出。我使用Linux或BSD电脑上工作时几乎都会使用它，并且我实际上为了使用 `KCachegrind` 而切换到Linux来编写代码。

教会你如何使用是这个练习之外的内容，你需要在这个练习之后自己学习如何用它。输出几乎是相同的，除了 `KCachegrind` 可以让你做这些：

- 图形化地浏览源码和执行次数，并使用各种排序来搜索可优化的东西。
- 分析不同的图表，来可视化地观察什么占据了大多数时间，以及它调用了什么。
- 查看真实的汇编机器码输出，使你能够看到实际的指令，给你更多的线索。
- 可视化地显示源码中的循环和分支的跳跃方式，便于你更容易地找到优化代码的方法。

你应该在获取、安装和玩转 `KCachegrind` 上花一些时间。

附加题

- 阅读 callgrind 手册页并且尝试一些高级选项。
- 阅读 cachegrind 手册页并且也尝试一些高级选项。
- 在所有单元测试上使用 `callgrind` 和 `cachegrind` ，看看你能否找到可优

化的地方。你找到一些预料之外的事情了吗？如果没有，你可能观察地不够仔细。

- 使用 KCachegrind 并且观察它和我这里的输出有什么不同。
- 现在使用这些工具来完成练习40的附加题和改进部分。

练习42：栈和队列

> 原文：Exercise 42: Stacks and Queues
>
> 译者：飞龙

到现在为止，你已经知道了大多数用于构建其它数据结构的数据结构。如果你拥有一些 `List` 、 `DArray` 、 `Hashmap` 和 `Tree` ，你就能用他们构造出大多数其它的任何结构。你碰到的其它任何结构要么可以用它们实现，要么是它们的变体。如果不是的话，它可能是外来的数据结构，你可能不需要它。

`Stack` 和 `Queue` 是非常简单的数据结构，它们是 `List` 的变体。它们是 `List` 的弱化或者转换形式，因为你只需要在 `List` 的一端放置元素。对于 `Stack` ，你只能能够在一段压入和弹出元素。而对于 `Queue` ，你只能够在开头压入元素，并在末尾弹出（或者反过来）。

我能够只通过C预处理器和两个头文件来实现这两个数据结构。我的头文件只有21行的长度，并且实现了所有 `Stack` 和 `Queue` 的操作，不带有任何神奇的定义。

我将会向你展示单元测试，你需要实现头文件来让它们正常工作。你不能创建 `stack.c` 或 `queue.c` 实现文件来通过测试，只能使用 `stack.h` 和 `queue.h` 来使测试运行。

```c
#include "minunit.h"
#include <lcthw/stack.h>
#include <assert.h>

static Stack *stack = NULL;
char *tests[] = {"test1 data", "test2 data", "test3 data"};
#define NUM_TESTS 3


char *test_create()
{
    stack = Stack_create();
    mu_assert(stack != NULL, "Failed to create stack.");

    return NULL;
}

char *test_destroy()
{
    mu_assert(stack != NULL, "Failed to make stack #2");
    Stack_destroy(stack);

    return NULL;
}

char *test_push_pop()
```

```c
{
    int i = 0;
    for(i = 0; i < NUM_TESTS; i++) {
        Stack_push(stack, tests[i]);
        mu_assert(Stack_peek(stack) == tests[i], "Wrong next val
ue.");
    }

    mu_assert(Stack_count(stack) == NUM_TESTS, "Wrong count on p
ush.");

    STACK_FOREACH(stack, cur) {
        debug("VAL: %s", (char *)cur->value);
    }

    for(i = NUM_TESTS - 1; i >= 0; i--) {
        char *val = Stack_pop(stack);
        mu_assert(val == tests[i], "Wrong value on pop.");
    }

    mu_assert(Stack_count(stack) == 0, "Wrong count after pop.")
;

    return NULL;
}

char *all_tests() {
    mu_suite_start();

    mu_run_test(test_create);
    mu_run_test(test_push_pop);
    mu_run_test(test_destroy);

    return NULL;
}

RUN_TESTS(all_tests);
```

之后是`queue_tests.c`，几乎以相同的方式来使用`Queue`：

```c
#include "minunit.h"
#include <lcthw/queue.h>
#include <assert.h>

static Queue *queue = NULL;
char *tests[] = {"test1 data", "test2 data", "test3 data"};
#define NUM_TESTS 3


char *test_create()
{
```

```
    queue = Queue_create();
    mu_assert(queue != NULL, "Failed to create queue.");

    return NULL;
}

char *test_destroy()
{
    mu_assert(queue != NULL, "Failed to make queue #2");
    Queue_destroy(queue);

    return NULL;
}

char *test_send_recv()
{
    int i = 0;
    for(i = 0; i < NUM_TESTS; i++) {
        Queue_send(queue, tests[i]);
        mu_assert(Queue_peek(queue) == tests[0], "Wrong next val
ue.");
    }

    mu_assert(Queue_count(queue) == NUM_TESTS, "Wrong count on s
end.");

    QUEUE_FOREACH(queue, cur) {
        debug("VAL: %s", (char *)cur->value);
    }

    for(i = 0; i < NUM_TESTS; i++) {
        char *val = Queue_recv(queue);
        mu_assert(val == tests[i], "Wrong value on recv.");
    }

    mu_assert(Queue_count(queue) == 0, "Wrong count after recv."
);

    return NULL;
}

char *all_tests() {
    mu_suite_start();

    mu_run_test(test_create);
    mu_run_test(test_send_recv);
    mu_run_test(test_destroy);

    return NULL;
}

RUN_TESTS(all_tests);
```

你应该在不修改测试文件的条件下，使单元测试能够运行，并且它应该能够通过 `valgrind` 而没有任何内存错误。下面是当我直接运行 `stack_tests` 时它的样子：

```
$ ./tests/stack_tests
DEBUG tests/stack_tests.c:60: ----- RUNNING: ./tests/stack_tests
----
RUNNING: ./tests/stack_tests
DEBUG tests/stack_tests.c:53:
----- test_create
DEBUG tests/stack_tests.c:54:
----- test_push_pop
DEBUG tests/stack_tests.c:37: VAL: test3 data
DEBUG tests/stack_tests.c:37: VAL: test2 data
DEBUG tests/stack_tests.c:37: VAL: test1 data
DEBUG tests/stack_tests.c:55:
----- test_destroy
ALL TESTS PASSED
Tests run: 3
$
```

`queue_test` 的输出基本一样，所以我在这里就不展示了。

如何改进

你可以做到的唯一真正的改进，就是把所用的 `List` 换成 `DArray` 。 `Queue` 数据结构难以用 `DArray` 实现，因为它要同时处理两端的节点。

完全在头文件中来实现它们的缺点，是你并不能够轻易地对它做性能调优。你需要使用这种技巧，建立一种以特定的方式使用 `List` 的“协议”。做性能调优时，如果你优化了 `List` ，这两种数据结构都会有所改进。

附加题

- 使用 `DArray` 代替 `List` 实现 `Stack` ，并保持单元测试不变。这意味着你需要创建你自己的 `STACK_FOREACH` 。

练习43：一个简单的统计引擎

> 原文：Exercise 43: A Simple Statistics Engine
>
> 译者：飞龙

这是一个简单的算法，我将其用于“联机”（不储存任何样本）收集概要统计。我在任何需要执行一些统计，比如均值、标准差和求和中使用它，但是其中我并不会储存所需的全部样本。我只需要储存计算出的结果，它们仅仅含有5个数值。

计算标准差和均值

首先你需要一系列样本。它可以使任何事情，比如完成一个任务所需的时间，某人访问某个东西的次数，或者甚至是网站的评分。是什么并不重要，只要你能得到一些数字，并且你想要知道它们的下列概要统计值：

`sum`

对所有数字求和。

`sumsq`（平方和）

对所有数字求平方和。

`count(n)`

求出样本数量。

`min`

求出样本最小值。

`max`

求出样本最大值。

`mean`

求出样本的均值。它类似于但又不是中位数，但可作为中位数的估计。

`stddev`

使用 `$sqrt(sumsq - (sum * mean) / (n - 1) )` 来计算标准差，其中 `sqrt` 为 `math.h` 头文件中的平方根。

我将会使用R来验证这些计算，因为我知道R能够计算正确。

```
> s <- runif(n=10, max=10)
> s
 [1] 6.1061334 9.6783204 1.2747090 8.2395131 0.3333483 6.9755066
1.0626275
 [8] 7.6587523 4.9382973 9.5788115
> summary(s)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.3333  2.1910  6.5410  5.5850  8.0940  9.6780
> sd(s)
[1] 3.547868
> sum(s)
[1] 55.84602
> sum(s * s)
[1] 425.1641
> sum(s) * mean(s)
[1] 311.8778
> sum(s * s) - sum(s) * mean(s)
[1] 113.2863
> (sum(s * s) - sum(s) * mean(s)) / (length(s) - 1)
[1] 12.58737
> sqrt((sum(s * s) - sum(s) * mean(s)) / (length(s) - 1))
[1] 3.547868
>
```

你并不需要懂得R，只需要看着我拆分代码来解释如何检查这些运算：

lines 1-4

我使用 `runit` 函数来获得“随机形式”的数字分布，之后将它们打印出来。我会在接下来的单元测试中用到它。

lines 5-7

这个就是概要，便于你看到R如何计算它们。

lines 8-9

这是使用 `sd` 函数计算的 `stddev` 。

lines 10-11

现在我开始手动进行这一计算，首先计算 `sum` 。

lines 12-13

`stddev` 公式中的下一部分是 `sumsq` ，我可以通过 `sum(s * s)` 来得到，它告诉R将整个 `s` 列表乘以其自身，之后计算它们的 `sum` 。R的可以在整个数据结构上做运算，就像这样。

lines 14-15

观察那个公式，我之后需要 `sum` 乘上 `mean` ，所以执行了 `sum(s) * mean(s)` 。

lines 16-17

我接着将 `sumsq` 参与运算，得到 `sum(s * s) - sum(s) * mean(s)` 。

lines 18-19

还需要除以 `n - 1` ，所以我执行了 `(sum(s * s) - sum(s) * mean(s)) / (length(s) - 1)` 。

lines 20-21

随后，我使用 `sqrt` 算出平方根，并得到3.547868，它符合R通过 `sd` 的运算结果。

实现

这就是计算 `stddev` 的方法，现在我可以编写一些简单的代码来实现这一计算。

```
#ifndef lcthw_stats_h
#define lctwh_stats_h

typedef struct Stats {
    double sum;
    double sumsq;
    unsigned long n;
    double min;
    double max;
} Stats;

Stats *Stats_recreate(double sum, double sumsq, unsigned long n,
double min, double max);

Stats *Stats_create();

double Stats_mean(Stats *st);

double Stats_stddev(Stats *st);

void Stats_sample(Stats *st, double s);

void Stats_dump(Stats *st);

#endif
```

这里你可以看到我将所需的统计量放入一个struct，并且创建了用于处理样本和获得数值的函数。实现它只是转换数字的一个练习：

```c
#include <math.h>
#include <lcthw/stats.h>
#include <stdlib.h>
#include <lcthw/dbg.h>

Stats *Stats_recreate(double sum, double sumsq, unsigned long n,
double min, double max)
{
    Stats *st = malloc(sizeof(Stats));
    check_mem(st);

    st->sum = sum;
    st->sumsq = sumsq;
    st->n = n;
    st->min = min;
    st->max = max;

    return st;

error:
    return NULL;
}

Stats *Stats_create()
{
    return Stats_recreate(0.0, 0.0, 0L, 0.0, 0.0);
}

double Stats_mean(Stats *st)
{
    return st->sum / st->n;
}

double Stats_stddev(Stats *st)
{
   return sqrt( (st->sumsq - ( st->sum * st->sum / st->n)) / (st
->n - 1) );
}

void Stats_sample(Stats *st, double s)
{
    st->sum += s;
    st->sumsq += s * s;

    if(st->n == 0) {
        st->min = s;
        st->max = s;
    } else {
        if(st->min > s) st->min = s;
        if(st->max < s) st->max = s;
    }
```

```c
    st->n += 1;
}

void Stats_dump(Stats *st)
{
    fprintf(stderr, "sum: %f, sumsq: %f, n: %ld, min: %f, max: %
f, mean: %f, stddev: %f",
            st->sum, st->sumsq, st->n, st->min, st->max,
            Stats_mean(st), Stats_stddev(st));
}
```

下面是 `stats.c` 中每个函数的作用：

Stats_recreate

我希望从一些数据中加载这些数据，这和函数让我重新创建 `Stats` 结构体。

Stats_create

只是以全0的值调用 `Stats_recreate` 。

Stats_mean

使用 `sum` 和 `n` 计算均值。

Stats_stddev

实现我之前的公式，唯一的不同就是我使用 `t->sum / st->n` 来计算均值，而不是调用 `Stats_mean` 。

Stats_sample

它用于在 `Stats` 结构体中储存数值。当你向它提供数值时，它看到 `n` 是0，并且相应地设置 `min` 和 `max` 。之后的每次调用都会使 `sum` 、 `sumsq` 和 `n` 增加，并且计算出这一新的样本的 `min` 和 `max` 值。

Stats_dump

简单的调试函数，用于转储统计量，便于你看到它们。

我需要干的最后一件事，就是确保这些运算正确。我打算使用我的样本，以及来自于R会话中的计算结果创建单元测试，来确保我会得到正确的结果。

```c
#include "minunit.h"
#include <lcthw/stats.h>
#include <math.h>

const int NUM_SAMPLES = 10;
double samples[] = {
    6.1061334, 9.6783204, 1.2747090, 8.2395131, 0.3333483,
    6.9755066, 1.0626275, 7.6587523, 4.9382973, 9.5788115
```

```
};

Stats expect = {
    .sumsq = 425.1641,
    .sum = 55.84602,
    .min = 0.333,
    .max = 9.678,
    .n = 10,
};
double expect_mean = 5.584602;
double expect_stddev = 3.547868;

#define EQ(X,Y,N) (round((X) * pow(10, N)) == round((Y) * pow(10
, N)))

char *test_operations()
{
    int i = 0;
    Stats *st = Stats_create();
    mu_assert(st != NULL, "Failed to create stats.");

    for(i = 0; i < NUM_SAMPLES; i++) {
        Stats_sample(st, samples[i]);
    }

    Stats_dump(st);

    mu_assert(EQ(st->sumsq, expect.sumsq, 3), "sumsq not valid")
;
    mu_assert(EQ(st->sum, expect.sum, 3), "sum not valid");
    mu_assert(EQ(st->min, expect.min, 3), "min not valid");
    mu_assert(EQ(st->max, expect.max, 3), "max not valid");
    mu_assert(EQ(st->n, expect.n, 3), "max not valid");
    mu_assert(EQ(expect_mean, Stats_mean(st), 3), "mean not vali
d");
    mu_assert(EQ(expect_stddev, Stats_stddev(st), 3), "stddev no
t valid");

    return NULL;
}

char *test_recreate()
{
    Stats *st = Stats_recreate(expect.sum, expect.sumsq, expect.
n, expect.min, expect.max);

    mu_assert(st->sum == expect.sum, "sum not equal");
    mu_assert(st->sumsq == expect.sumsq, "sumsq not equal");
    mu_assert(st->n == expect.n, "n not equal");
    mu_assert(st->min == expect.min, "min not equal");
    mu_assert(st->max == expect.max, "max not equal");
    mu_assert(EQ(expect_mean, Stats_mean(st), 3), "mean not vali
d");
```

```
    mu_assert(EQ(expect_stddev, Stats_stddev(st), 3), "stddev no
t valid");

    return NULL;
}

char *all_tests()
{
    mu_suite_start();

    mu_run_test(test_operations);
    mu_run_test(test_recreate);

    return NULL;
}

RUN_TESTS(all_tests);
```

这个单元测试中没什么新东西，除了 `EQ` 宏。我比较懒，并且不想查询比较两个 `double` 值的标准方法，所以我使用了这个宏。 `double` 的问题是等性不是完全相等，因为我使用了两个不同的系统，并带有不同的四舍五入的位数。解决方案就是判断两个数“乘以10的X次方是否相等”。

我使用 `EQ` 来计算数字的10的幂，之后使用 `round` 函数来获得证书。这是个简单的方法来四舍五入N位小数，并以整数比较结果。我确定有数以亿计的其它方法能做相同的事情，但是现在我就用这种。

预期结果储存在 `Stats` `struct` 中，之后我只是确保我得到的数值接近R给我的数值。

如何使用

你可以使用标准差和均值来决定一个新的样本是否是“有趣”的，或者你可以使用它们计算统计量的统计量。前者对于人们来说更容易理解，所以我用登录的例子来做个简短的解释。

假设你在跟踪人们花费多长时间在一台服务器上，并且你打算用统计来分析它。每次有人登录进来，你都对它们在这里的时长保持跟踪，之后调用 `Stats_sample` 函数。我会寻找停留“过长”时间的人，以及“过短”的人。

比起设定特殊的级别，我更倾向于将一个人的停留时间与 `mean (plus or minus) 2 * stddev` 这个范围进行比较。我计算出 `mean` 和 `2 * stddev` ，并且如果它们在这个范围之外，我就认为是“有趣”的。由于我使用了联机算法来维护这些统计量，所以它非常快，并且我可以使软件标记在这个范围外的用户。

这不仅仅用于找出行为异常的用户，更有助于标记一些潜在的问题，你可以查看它们来观察发生了什么。它基于所有用户的行为来计算，这也避免了你任意挑出一个数值而并不基于实际情况的问题。

你可以从中学到的通用规则是， `mean (plus or minus) 2 * stddev` 是90%的值预期所属的范围预测值，任何在它之外的值都是有趣的。

第二种利用这些统计量的方式就是继续将其用于其它的 `Stats` 计算。基本上像通常一样使用 `Stats_sample` ，但是之后在 `min` 、 `max` 、 `n` 、 `mean` 和 `stddev` 上执行 `Stats_sample` 。这会提供二级的度量，并且让你对比样本的样本。

被搞晕了吗？我会以上面的例子基础，并且假设你拥有100台服务器，每台都运行一个应用。你已经在每个应用服务器上跟踪了用户的登录时长，但是你想要比较所有的这100和应用，并且标记它们当中任何登录时间过长的用户。最简单的方式就是每次有人登录进来时，计算新的登录统计量，之后将 `Stats structs` 的元素添加到第二个 `Stats` 中。

你最后应该会得到一些统计量，它们可以这样命名：

均值的均值

这是一个 `Stats struct` ，它向你提供所有服务器的均值的 `mean` 和 `stddev` 。你可以用全局视角来观察任何在此之外的用户或服务器。

标准差的均值

另一个 `Stats struct` ，计算这些服务器的分布的统计量。你之后可以分析每个服务器并且观察是否它们中的任何服务器具有异常分散的分布，通过将它们的 `stddev` 和这个 `mean of stddevs` 统计量进行对比。

你可以计算出全部统计量，但是这两个是最有用的。如果你打算监视服务器上的移除登录时间，你可以这样做：

- 用户John登录并登出服务器A。获取服务器A的统计量，并更新它们。
- 获取 `mean of means` 统计量，计算出A的均值并且将其加入样本。我叫它 `m_of_m` 。
- 获取 `mean of stddev` 统计量，将A的标准差添加到样本中。我叫它 `m_of_s` 。
- 如果A的 `mean` 在 `m_of_m.mean + 2 * m_of_m.stddev` 范围外，标记它可能存在问题。
- 如果A的 `stddev` 在 `m_of_s.mean + 2 * m_of_s.stddev` 范围外，标记它可能存在行为异常。
- 最后，如果John的登录时长在A的范围之外，或A的 `m_of_m` 范围之外，标记为有趣的。

通过计算“均值的均值”，或者“标准差的均值”，你可以以最小的执行和储存总量，有效地跟踪许多度量。

附加题

- 将 `Stats_stddev` 和 `Stats_mean` 转换为 `static inline` 函数，放到 `stats.h` 文件中，而不是 `stats.c` 文件。

- 使用这份代码来编写 `string_algos_test.c` 的性能测试。使它为可选的，并且运行基准测试作为一系列样本，之后报告结果。
- 编写它的另一个语言的版本。确保这个版本基于我的数据正确执行。
- 编写一个小型程序，它能从文件读取所有数字，并执行这些统计。
- 使程序接收一个数据表，其中第一行是表头，剩下的行含有任意数量空格分隔的数值。你的程序应该按照表头中的名称，打印出每一列的统计值。

练习44：环形缓冲区

> 原文：Exercise 44: Ring Buffer
>
> 译者：飞龙

环形缓冲区在处理异步IO时非常实用。它们可以在一端接收随机长度和区间的数据，在另一端以相同长度和区间提供密致的数据块。它们是 `Queue` 数据结构的变体，但是它针对于字节块而不是一系列指针。这个练习中我打算向你展示 `RingBuffer` 的代码，并且之后你需要对它执行完整的单元测试。

```
#ifndef _lcthw_RingBuffer_h
#define _lcthw_RingBuffer_h

#include <lcthw/bstrlib.h>

typedef struct {
    char *buffer;
    int length;
    int start;
    int end;
} RingBuffer;

RingBuffer *RingBuffer_create(int length);

void RingBuffer_destroy(RingBuffer *buffer);

int RingBuffer_read(RingBuffer *buffer, char *target, int amount)
;

int RingBuffer_write(RingBuffer *buffer, char *data, int length)
;

int RingBuffer_empty(RingBuffer *buffer);

int RingBuffer_full(RingBuffer *buffer);

int RingBuffer_available_data(RingBuffer *buffer);

int RingBuffer_available_space(RingBuffer *buffer);

bstring RingBuffer_gets(RingBuffer *buffer, int amount);

#define RingBuffer_available_data(B) (((B)->end + 1) % (B)->leng
th - (B)->start - 1)

#define RingBuffer_available_space(B) ((B)->length - (B)->end -
1)
```

```
#define RingBuffer_full(B) (RingBuffer_available_data((B)) - (B)
->length == 0)

#define RingBuffer_empty(B) (RingBuffer_available_data((B)) == 0)


#define RingBuffer_puts(B, D) RingBuffer_write((B), bdata((D)),
blength((D)))

#define RingBuffer_get_all(B) RingBuffer_gets((B), RingBuffer_av
ailable_data((B)))

#define RingBuffer_starts_at(B) ((B)->buffer + (B)->start)

#define RingBuffer_ends_at(B) ((B)->buffer + (B)->end)

#define RingBuffer_commit_read(B, A) ((B)->start = ((B)->start +
 (A)) % (B)->length)

#define RingBuffer_commit_write(B, A) ((B)->end = ((B)->end + (A
)) % (B)->length)

#endif
```

观察这个数据结构，你会看到它含有 `buffer` 、 `start` 和 `end` 。 `RingBuffer` 的所做的事情只是在 `buffer` 中移动 `start` 和 `end` ，所以当数据到达缓冲区末尾时还可以继续“循环”。这样就会给人一种在固定空间内无限读取的“幻觉”。接下来我创建了一些宏来基于它执行各种计算。

下面是它的实现，它是对工作原理更好的解释：

```
#undef NDEBUG
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <lcthw/dbg.h>
#include <lcthw/ringbuffer.h>

RingBuffer *RingBuffer_create(int length)
{
    RingBuffer *buffer = calloc(1, sizeof(RingBuffer));
    buffer->length  = length + 1;
    buffer->start = 0;
    buffer->end = 0;
    buffer->buffer = calloc(buffer->length, 1);

    return buffer;
}
```

```
void RingBuffer_destroy(RingBuffer *buffer)
{
    if(buffer) {
        free(buffer->buffer);
        free(buffer);
    }
}

int RingBuffer_write(RingBuffer *buffer, char *data, int length)
{
    if(RingBuffer_available_data(buffer) == 0) {
        buffer->start = buffer->end = 0;
    }

    check(length <= RingBuffer_available_space(buffer),
            "Not enough space: %d request, %d available",
            RingBuffer_available_data(buffer), length);

    void *result = memcpy(RingBuffer_ends_at(buffer), data, leng
th);
    check(result != NULL, "Failed to write data into buffer.");

    RingBuffer_commit_write(buffer, length);

    return length;
error:
    return -1;
}

int RingBuffer_read(RingBuffer *buffer, char *target, int amount)

{
    check_debug(amount <= RingBuffer_available_data(buffer),
            "Not enough in the buffer: has %d, needs %d",
            RingBuffer_available_data(buffer), amount);

    void *result = memcpy(target, RingBuffer_starts_at(buffer),
amount);
    check(result != NULL, "Failed to write buffer into data.");

    RingBuffer_commit_read(buffer, amount);

    if(buffer->end == buffer->start) {
        buffer->start = buffer->end = 0;
    }

    return amount;
error:
    return -1;
}

bstring RingBuffer_gets(RingBuffer *buffer, int amount)
{
```

```
    check(amount > 0, "Need more than 0 for gets, you gave: %d "
, amount);
    check_debug(amount <= RingBuffer_available_data(buffer),
            "Not enough in the buffer.");

    bstring result = blk2bstr(RingBuffer_starts_at(buffer), amou
nt);
    check(result != NULL, "Failed to create gets result.");
    check(blength(result) == amount, "Wrong result length.");

    RingBuffer_commit_read(buffer, amount);
    assert(RingBuffer_available_data(buffer) >= 0 && "Error in r
ead commit.");

    return result;
error:
    return NULL;
}
```

这些就是一个基本的 `RingBuffer` 实现的全部了。你可以从中读取和写入数据，获得它的大小和容量。也有一些缓冲区使用OS中的技巧来创建虚拟的无限存储，但它们不可移植。

由于我的 `RingBuffer` 处理读取和写入内存块，我要保证任何 `end == start` 出现的时候我都要将它们重置为0，使它们从退回缓冲区头部。在维基百科上的版本中，它并不可以写入数据块，所以只能移动 `end` 和 `start` 来转圈。为了更好地处理数据块，你需要在数据为空时移动到内部缓冲区的开头。

单元测试

对于你的单元测试，你需要测试尽可能多的情况。最简单的方法就是预构造不同的 `RingBuffer` 结构，之后手动检查函数和算数是否有效。例如，你可以构造 `end` 在缓冲区末尾的右边，而 `start` 在缓冲区范围内的 `RingBuffer` ，来看看它是否执行成功。

你会看到什么

下面是我的 `ringbuffer_tests` 运行结果：

```
$ ./tests/ringbuffer_tests
DEBUG tests/ringbuffer_tests.c:60: ----- RUNNING: ./tests/ringbu
ffer_tests
----
RUNNING: ./tests/ringbuffer_tests
DEBUG tests/ringbuffer_tests.c:53:
----- test_create
DEBUG tests/ringbuffer_tests.c:54:
----- test_read_write
DEBUG tests/ringbuffer_tests.c:55:
----- test_destroy
ALL TESTS PASSED
Tests run: 3
$
```

你应该测试至少三次来确保所有基本操作有效，并且看看在我完成之前你能测试到额外的多少东西。

如何改进

像往常一样，你应该为这个练习做防御性编程检查。我希望你这样做，是因为 `liblcthw` 的代码基本上没有做我教给你的防御型编程检查。我将它们留给你，便于你熟悉使用这些额外的检查来改进代码。

例如，这个环形缓冲区并没有过多检查每次访问是否实际上都在缓冲区内。

如果你阅读环形缓冲区的维基百科页面，你会看到“优化的POSIX实现”，它使用POSIX特定的调用来创建一块无限的区域。研究并且在附加题中尝试实现它。

附加题

- 创建 `RingBuffer` 的替代版本，使用POSIX的技巧并为其执行单元测试。
- 为二者添加一个性能对比测试，通过带有随机数据和随机读写操作的模糊测试来比较两个版本。确保你你对每个版本进行了相同的操作，便于你在操作之间比较二者。
- 使用 `callgrind` 和 `cachegrind` 比较二者的性能。

练习45：一个简单的TCP/IP客户端

> 原文：Exercise 45: A Simple TCP/IP Client
>
> 译者：飞龙

我打算使用 `RingBuffer` 来创建一个非常简单的小型网络测试工具，叫做 `netclient` 。为此我需要向 `Makefile` 添加一些工具，来处理 `bin/` 目录下的小程序。

扩展Makefile

首先，为程序添加一些变量，就像单元测试的 `TESTS` 和 `TEST_SRC` 变量：

```
PROGRAMS_SRC=$(wildcard bin/*.c)
PROGRAMS=$(patsubst %.c,%,$(PROGRAMS_SRC))
```

之后你可能想要添加 `PROGRAMS` 到所有目标中：

```
all: $(TARGET) $(SO_TARGET) tests $(PROGRAMS)
```

之后在 `clean` 目标中向 `rm` 那一行添加 `PROGRAMS` ：

```
rm -rf build $(OBJECTS) $(TESTS) $(PROGRAMS)
```

最后你还需要在最后添加一个目标来构建它们：

```
$(PROGRAMS): CFLAGS += $(TARGET)
```

做了这些修改你就能够将 `.c` 文件扔到 `bin` 中，并且编译它们以及为其链接库文件，就像测试那样。

netclient 代码

netclient的代码是这样的：

```
#undef NDEBUG
#include <stdlib.h>
#include <sys/select.h>
```

```c
#include <stdio.h>
#include <lcthw/ringbuffer.h>
#include <lcthw/dbg.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <fcntl.h>

struct tagbstring NL = bsStatic("\n");
struct tagbstring CRLF = bsStatic("\r\n");

int nonblock(int fd) {
    int flags = fcntl(fd, F_GETFL, 0);
    check(flags >= 0, "Invalid flags on nonblock.");

    int rc = fcntl(fd, F_SETFL, flags | O_NONBLOCK);
    check(rc == 0, "Can't set nonblocking.");

    return 0;
error:
    return -1;
}

int client_connect(char *host, char *port)
{
    int rc = 0;
    struct addrinfo *addr = NULL;

    rc = getaddrinfo(host, port, NULL, &addr);
    check(rc == 0, "Failed to lookup %s:%s", host, port);

    int sock = socket(AF_INET, SOCK_STREAM, 0);
    check(sock >= 0, "Cannot create a socket.");

    rc = connect(sock, addr->ai_addr, addr->ai_addrlen);
    check(rc == 0, "Connect failed.");

    rc = nonblock(sock);
    check(rc == 0, "Can't set nonblocking.");

    freeaddrinfo(addr);
    return sock;

error:
    freeaddrinfo(addr);
    return -1;
}

int read_some(RingBuffer *buffer, int fd, int is_socket)
{
```

```
    int rc = 0;

    if(RingBuffer_available_data(buffer) == 0) {
        buffer->start = buffer->end = 0;
    }

    if(is_socket) {
        rc = recv(fd, RingBuffer_starts_at(buffer), RingBuffer_a
vailable_space(buffer), 0);
    } else {
        rc = read(fd, RingBuffer_starts_at(buffer), RingBuffer_a
vailable_space(buffer));
    }

    check(rc >= 0, "Failed to read from fd: %d", fd);

    RingBuffer_commit_write(buffer, rc);

    return rc;

error:
    return -1;
}


int write_some(RingBuffer *buffer, int fd, int is_socket)
{
    int rc = 0;
    bstring data = RingBuffer_get_all(buffer);

    check(data != NULL, "Failed to get from the buffer.");
    check(bfindreplace(data, &NL, &CRLF, 0) == BSTR_OK, "Failed
to replace NL.");

    if(is_socket) {
        rc = send(fd, bdata(data), blength(data), 0);
    } else {
        rc = write(fd, bdata(data), blength(data));
    }

    check(rc == blength(data), "Failed to write everything to fd
: %d.", fd);
    bdestroy(data);

    return rc;

error:
    return -1;
}


int main(int argc, char *argv[])
{
```

```
    fd_set allreads;
    fd_set readmask;

    int socket = 0;
    int rc = 0;
    RingBuffer *in_rb = RingBuffer_create(1024 * 10);
    RingBuffer *sock_rb = RingBuffer_create(1024 * 10);

    check(argc == 3, "USAGE: netclient host port");

    socket = client_connect(argv[1], argv[2]);
    check(socket >= 0, "connect to %s:%s failed.", argv[1], argv[
2]);

    FD_ZERO(&allreads);
    FD_SET(socket, &allreads);
    FD_SET(0, &allreads);

    while(1) {
        readmask = allreads;
        rc = select(socket + 1, &readmask, NULL, NULL, NULL);
        check(rc >= 0, "select failed.");

        if(FD_ISSET(0, &readmask)) {
            rc = read_some(in_rb, 0, 0);
            check_debug(rc != -1, "Failed to read from stdin.");
        }

        if(FD_ISSET(socket, &readmask)) {
            rc = read_some(sock_rb, socket, 0);
            check_debug(rc != -1, "Failed to read from socket.")
;
        }

        while(!RingBuffer_empty(sock_rb)) {
            rc = write_some(sock_rb, 1, 0);
            check_debug(rc != -1, "Failed to write to stdout.");
        }

        while(!RingBuffer_empty(in_rb)) {
            rc = write_some(in_rb, socket, 1);
            check_debug(rc != -1, "Failed to write to socket.");
        }
    }

    return 0;

error:
    return -1;
}
```

代码中使用了`select`来处理`stdin`（文件描述符0）和用于和服务器交互的`socket`中的事件。它使用了`RingBuffer`来储存和复制数据，并且你可以认为`read_some`和`write_some`函数都是`RingBuffer`中相似函数的原型。

在这一小段代码中，可能有一些你并不知道的网络函数。当你碰到不知道的函数时，在手册页上查询它来确保你理解了它。这一小段代码可能需要让你研究用于小型服务器编程的所有C语言API。

你会看到什么

如果你完成了所有构建，测试的最快方式就是看看你能否从learncodethehardway.org上得到一个特殊的文件：

```
$
$ ./bin/netclient learncodethehardway.org 80
GET /ex45.txt HTTP/1.1
Host: learncodethehardway.org

HTTP/1.1 200 OK
Date: Fri, 27 Apr 2012 00:41:25 GMT
Content-Type: text/plain
Content-Length: 41
Last-Modified: Fri, 27 Apr 2012 00:42:11 GMT
ETag: 4f99eb63-29
Server: Mongrel2/1.7.5

Learn C The Hard Way, Exercise 45 works.
^C
$
```

这里我所做的事情是键入创建`/ex45.txt`的HTTP请求所需的语法，在`Host:`请求航之后，按下ENTER键来输入空行。接着我获取相应，包括响应头和内容。最后我按下CTRL-C来退出。

如何使它崩溃

这段代码肯定含有bug，但是当前在本书的草稿中，我会继续完成它。与此同时，尝试分析代码，并且用其它服务器来击溃它。一种叫做`netcat`的工具可以用于建立这种服务器。另一种方法就是使用`Python`或`Ruby`之类的语言创建一个简单的“垃圾服务器”，来产生垃圾数据，随机关闭连接，或者其它异常行为。

如果你找到了bug，在评论中报告它们，我会修复它。

附加题

- 像我提到的那样，这里面有一些你不知道的函数，去查询他们。实际上，即使你知道它们也要查询。
- 在 `valgrind` 下运行它来寻找错误。
- 为函数添加各种防御性编程检查，来改进它们。
- 使用 `getopt` 函数，运行用户提供选项来防止将 `\n` 转换为 `\r\n` 。这仅仅用于需要处理行尾的协议例如HTTP。有时你可能不想执行转换，所以要给用户一个选择。

练习46：三叉搜索树

> 原文：Exercise 46: Ternary Search Tree
>
> 译者：飞龙

我打算向你介绍的最后一种数据结构就是三叉搜索树（ `TSTree` ），它和 `BSTree` 很像，除了它有三个分支， `low` 、 `equal` 和 `high` 。它的用法和 `BStree` 以及 `Hashmap` 基本相同，用于储存键值对的数据，但是它通过键中的独立字符来控制。这使得 `TSTree` 具有一些 `BStree` 和 `Hashmap` 不具备的功能。

`TSTree` 的工作方式是，每个键都是字符串，根据字符串中字符的等性，通过构建或者遍历一棵树来进行插入。首先由根节点开始，观察每个节点的字符，如果小于、等于或大于则去往相应的方向。你可以参考这个头文件：

```c
#ifndef _lcthw_TSTree_h
#define _lctwh_TSTree_h

#include <stdlib.h>
#include <lcthw/darray.h>

typedef struct TSTree {
    char splitchar;
    struct TSTree *low;
    struct TSTree *equal;
    struct TSTree *high;
    void *value;
} TSTree;

void *TSTree_search(TSTree *root, const char *key, size_t len);

void *TSTree_search_prefix(TSTree *root, const char *key, size_t
 len);

typedef void (*TSTree_traverse_cb)(void *value, void *data);

TSTree *TSTree_insert(TSTree *node, const char *key, size_t len,
void *value);

void TSTree_traverse(TSTree *node, TSTree_traverse_cb cb, void *
data);

void TSTree_destroy(TSTree *root);

#endif
```

`TSTree` 拥有下列成员：

splitchar

树中该节点的字符。

low

小于 `splitchar` 的分支。

equal

等于 `splitchar` 的分支。

high

大于 `splitchar` 的分支。

value

这个节点上符合当前 `splitchar` 的值的集合。

你可以看到这个实现中含有下列操作：

search

为特定 `key` 寻找值的典型操作。

search_prefix

寻找第一个以 `key` 为前缀的值，这是你不能轻易使用 `BSTree` 或 `Hashmap` 完成的操作。

insert

将 `key` 根据每个字符拆分，并把它插入到树中。

traverse

遍历整颗树，使你能够收集或分析所包含的所有键和值。

唯一缺少的操作就是 `TSTree_delete` ，这是因为它是一个开销很大的操作，比 `BSTree_delete` 大得多。当我使用 `TSTree` 结构时，我将它们视为常量数据，我打算遍历许多次，但是永远不会移除任何东西。它们对于这样的操作会很快，但是不适于需要快速插入或删除的情况。为此我会使用 `Hashmap` 因为它优于 `BSTree` 和 `TSTree` 。

`TSTree` 的实现非常简单，但是第一次可能难以理解。我会在你读完之后拆分它。

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <lcthw/dbg.h>
#include <lcthw/tstree.h>

static inline TSTree *TSTree_insert_base(TSTree *root, TSTree *n
```

```
ode,
        const char *key, size_t len, void *value)
{
    if(node == NULL) {
        node = (TSTree *) calloc(1, sizeof(TSTree));

        if(root == NULL) {
            root = node;
        }

        node->splitchar = *key;
    }

    if(*key < node->splitchar) {
        node->low = TSTree_insert_base(root, node->low, key, len
, value);
    } else if(*key == node->splitchar) {
        if(len > 1) {
            node->equal = TSTree_insert_base(root, node->equal,
key+1, len - 1, value);
        } else {
            assert(node->value == NULL && "Duplicate insert into
 tst.");
            node->value = value;
        }
    } else {
        node->high = TSTree_insert_base(root, node->high, key, l
en, value);
    }

    return node;
}

TSTree *TSTree_insert(TSTree *node, const char *key, size_t len,
void *value)
{
    return TSTree_insert_base(node, node, key, len, value);
}

void *TSTree_search(TSTree *root, const char *key, size_t len)
{
    TSTree *node = root;
    size_t i = 0;

    while(i < len && node) {
        if(key[i] < node->splitchar) {
            node = node->low;
        } else if(key[i] == node->splitchar) {
            i++;
            if(i < len) node = node->equal;
        } else {
            node = node->high;
        }
```

```c
    }

    if(node) {
        return node->value;
    } else {
        return NULL;
    }
}

void *TSTree_search_prefix(TSTree *root, const char *key, size_t
 len)
{
    if(len == 0) return NULL;

    TSTree *node = root;
    TSTree *last = NULL;
    size_t i = 0;

    while(i < len && node) {
        if(key[i] < node->splitchar) {
            node = node->low;
        } else if(key[i] == node->splitchar) {
            i++;
            if(i < len) {
                if(node->value) last = node;
                node = node->equal;
            }
        } else {
            node = node->high;
        }
    }

    node = node ? node : last;

    // traverse until we find the first value in the equal chain
    // this is then the first node with this prefix
    while(node && !node->value) {
        node = node->equal;
    }

    return node ? node->value : NULL;
}

void TSTree_traverse(TSTree *node, TSTree_traverse_cb cb, void *
data)
{
    if(!node) return;

    if(node->low) TSTree_traverse(node->low, cb, data);

    if(node->equal) {
        TSTree_traverse(node->equal, cb, data);
    }
```

```c
    if(node->high) TSTree_traverse(node->high, cb, data);

    if(node->value) cb(node->value, data);
}

void TSTree_destroy(TSTree *node)
{
    if(node == NULL) return;

    if(node->low) TSTree_destroy(node->low);

    if(node->equal) {
        TSTree_destroy(node->equal);
    }

    if(node->high) TSTree_destroy(node->high);

    free(node);
}
```

对于 `TSTree_insert` ，我使用了相同模式的递归结构，其中我创建了一个小型函数，它调用真正的递归函数。我对此并不做任何检查，但是你应该为之添加通常的防御性编程策略。要记住的一件事，就是它使用了一些不同的设计，这里并没有单独的 `TSTree_create` 函数，如果你将 `node` 传入为 `NULL` ，它会新建一个，然后返回最终的值。

这意味着我需要为你分解 `TSTree_insert_base` ，使你理解插入操作。

tstree.c:10-18

像我提到的那样，如果函数接收到 `NULL` ，我需要创建节点，并且将 `*key` （当前字符）赋值给它。这用于当我插入键时来构建树。

tstree.c:20-21

当 `*key` 小于 `splitchar` 时，选择 `low` 分支。

tstree.c:22

如果 `splitchar` 相等，我就要进一步确定等性。这会在我刚刚创建这个节点时发生，所以这里我会构建这棵树。

tstree.c:23-24

仍然有字符串需要处理，所以向下递归 `equal` 分支，并且移动到下一个 `*key` 字符。

tstree.c:26-27

这是最后一个字符的情况，所以我将值设置好。我编写了一个`assert`来避免重复。

tstree.c:29-30

最后的情况是`*key`大于`splitchar`，所以我需要向下递归`high`分支。

这个数据结构的`key`实际上带有一些特性，我只会在`splitchar`相等时递增所要分析的字符。其它两种情况我只会继续遍历整个树，直到碰到了相等的字符，我才会递归处理下一个字符。这一操作使它对于找不到键的情况是非常快的。我可以传入一个不存在的键，简单地遍历一些`high`和`low`节点，直到我碰到了末尾并且知道这个键不存在。我并不需要处理键的每个字符，或者树的每个节点。

一旦你理解了这些，之后来分析`TSTree_search`如何工作：

tstree.c:46

我并不需要递归处理整棵树，只需要使用`while`循环和当前的`node`节点。

tstree.c:47-48

如果当前字符小于节点中的`splitchar`，则选择`low`分支。

tstree.c:49-51

如果相等，自增`i`并且选择`equal`分支，只要不是最后一个字符。这就是`if(i < len)`所做的，使我不会越过最后的`value`。

tstree.c:52-53

否则我会选择`high`分支，由于当前字符更大。

tstree.c:57-61

循环结束后如果`node`不为空，那么返回它的`value`，否则返回`NULL`。

这并不难以理解，并且你可以看到`TSTree_search_prefix`函数用了几乎相同的算法。唯一的不同就是我并不试着寻找精确的匹配，而是可找到的最长前缀。我在相等时跟踪`last`节点来实现它，并且在搜索循环结束之后，遍历这个节点直到发现`value`。

观察`TSTree_search_prefix`，你就会开始明白`TSTree`相对`BSTree`和`Hashmap`在查找操作上的另一个优点。给定一个长度为X的键，你可以在X步内找到任何键，但是也可以在X步加上额外的N步内找到第一个前缀，取决于匹配的键有多长。如果树中最长的键是十个字符，那么你就可以在10步之内找到任意的前缀。更重要的是，你可以通过对键的每个字符只比较一次来实现。

相比之下，使用`BSTree`执行相同操作，你需要在`BSTree`的每一个可能匹配的节点中检查两个字符串是否有共同的前缀。这对于寻找键，或者检查键是否存在（`TSTree_search`）是相同的。你需要将每个字符与`BSTree`中的大多数字符对比，来确认是否匹配。

`Hashamp` 对于寻找前缀更加糟糕，因为你不能够仅仅计算前缀的哈希值。你基本上不能高效在 `Hashmap` 中实现它，除非数据类似URL可以被解析。即使这样你还是需要遍历 `Hashmap` 的所有节点。

> 译者注：二叉树和三叉树在搜索时都是走其中的一支，但由于二叉树中每个节点储存字符串，而三叉树储存的是字符。所以三叉树的整个搜索过程相当于一次字符串比较，而二叉树的每个节点都需要一次字符串比较。三叉树堆叠储存字符串使搜索起来更方便。
>
> 至于哈希表，由于字符串整体和前缀计算出来的哈希值差别很大，所以按前缀搜索时，哈希的优势完全失效，所以只能改为暴力搜索，效果比二叉树还要差。

最后的两个函数应该易于分析，因为它们是典型的遍历和销毁操作，你已经在其它数据结构中看到过了。

最后，我编写了简单的单元测试，来确保我所做的全部东西正确。

```c
#include "minunit.h"
#include <lcthw/tstree.h>
#include <string.h>
#include <assert.h>
#include <lcthw/bstrlib.h>


TSTree *node = NULL;
char *valueA = "VALUEA";
char *valueB = "VALUEB";
char *value2 = "VALUE2";
char *value4 = "VALUE4";
char *reverse = "VALUER";
int traverse_count = 0;

struct tagbstring test1 = bsStatic("TEST");
struct tagbstring test2 = bsStatic("TEST2");
struct tagbstring test3 = bsStatic("TSET");
struct tagbstring test4 = bsStatic("T");

char *test_insert()
{
    node = TSTree_insert(node, bdata(&test1), blength(&test1), v
alueA);
    mu_assert(node != NULL, "Failed to insert into tst.");

    node = TSTree_insert(node, bdata(&test2), blength(&test2), v
alue2);
    mu_assert(node != NULL, "Failed to insert into tst with seco
nd name.");

    node = TSTree_insert(node, bdata(&test3), blength(&test3), r
everse);
    mu_assert(node != NULL, "Failed to insert into tst with reve
```

```
rse name.");

    node = TSTree_insert(node, bdata(&test4), blength(&test4), v
alue4);
    mu_assert(node != NULL, "Failed to insert into tst with seco
nd name.");

    return NULL;
}

char *test_search_exact()
{
    // tst returns the last one inserted
    void *res = TSTree_search(node, bdata(&test1), blength(&test
1));
    mu_assert(res == valueA, "Got the wrong value back, should g
et A not B.");

    // tst does not find if not exact
    res = TSTree_search(node, "TESTNO", strlen("TESTNO"));
    mu_assert(res == NULL, "Should not find anything.");

    return NULL;
}

char *test_search_prefix()
{
    void *res = TSTree_search_prefix(node, bdata(&test1), blengt
h(&test1));
    debug("result: %p, expected: %p", res, valueA);
    mu_assert(res == valueA, "Got wrong valueA by prefix.");

    res = TSTree_search_prefix(node, bdata(&test1), 1);
    debug("result: %p, expected: %p", res, valueA);
    mu_assert(res == value4, "Got wrong value4 for prefix of 1."
);

    res = TSTree_search_prefix(node, "TE", strlen("TE"));
    mu_assert(res != NULL, "Should find for short prefix.");

    res = TSTree_search_prefix(node, "TE--", strlen("TE--"));
    mu_assert(res != NULL, "Should find for partial prefix.");


    return NULL;
}

void TSTree_traverse_test_cb(void *value, void *data)
{
    assert(value != NULL && "Should not get NULL value.");
    assert(data == valueA && "Expecting valueA as the data.");
    traverse_count++;
}
```

```c
char *test_traverse()
{
    traverse_count = 0;
    TSTree_traverse(node, TSTree_traverse_test_cb, valueA);
    debug("traverse count is: %d", traverse_count);
    mu_assert(traverse_count == 4, "Didn't find 4 keys.");

    return NULL;
}

char *test_destroy()
{
    TSTree_destroy(node);

    return NULL;
}

char * all_tests() {
    mu_suite_start();

    mu_run_test(test_insert);
    mu_run_test(test_search_exact);
    mu_run_test(test_search_prefix);
    mu_run_test(test_traverse);
    mu_run_test(test_destroy);

    return NULL;
}

RUN_TESTS(all_tests);
```

优点和缺点

`TSTree` 可以用于实现一些其它实用的事情：

- 除了寻找前缀，你可以反转插入的所有键，之后通过后缀来寻找。我使用它来寻找主机名称，因为我想要找到 `*.learncodethehardway.com` ，所以如果我反向来寻找，会更快匹配到它们。
- 你可以执行“模糊”搜索，其中你可以收集所有与键的大多数字符相似的节点，或者使用其它算法用于搜索近似的匹配。
- 你可以寻找所有中间带有特定部分的键。

我已经谈论了 `TSTree` 能做的一些事情，但是它们并不总是最好的数据结构。 `TSTree` 的缺点在于：

- 像我提到过的那样，删除操作非常麻烦。它们适用于需要快速检索并且从不移除的操作。如果你需要删除，可以简单地将 `value` 置空，之后当树过大时周期性重构它。
- 与 `BSTree` 和 `Hashmap` 相比，它在相同的键上使用了大量的空间。它对于键

中的每个字符都使用了完整的节点。它对于短的键效果更好，但如果你在 `TSTree` 中放入一大堆东西，它会变得很大。
- 它们也不适合处理非常长的键，然而“长”是主观的词，所以应当像通常一样先进行测试。如果你尝试储存一万个字符的键，那么应当使用 `Hashmap` 。

如何改进

像通常一样，浏览代码，使用防御性的先决条件、断言，并且检查每个函数来改进。下面是一些其他的改进方案，但是你并不需要全部实现它们：

- 你可以使用 `DArray` 来允许重复的 `value` 值。
- 因为我提到删除非常困难，但是你可以通过将值设为 `NULL` 来模拟，使值能够高效被删除。
- 目前还不能获取到所有匹配指定前缀的值，我会让你在附加题中实现它。
- 有一些其他得更复杂的算法会比它要好。查询前缀数组、前缀树和基数树的资料。

附加题

- 实现 `TSTree_collect` 返回 `DArray` 包含所有匹配指定前缀的键。
- 实现 `TSTree_search_suffix` 和 `TSTree_insert_suffix` ，实现后缀搜索和插入。
- 使用 `valgrind` 来查看与 `BSTree` 和 `Hashmap` 相比，这个结构使用了多少内存来储存数据。

练习47：一个快速的URL路由

> 原文：Exercise 47: A Fast URL Router
>
> 译者：飞龙

我现在打算向你展示使用`TSTree`来创建服务器中的快速URL路由。它适用于应用中的简单的URL匹配，而不是在许多Web应用框架中的更复杂（一些情况下也不必要）的路由发现功能。

我打算编程一个小型命令行工具和路由交互，他叫做`urlor`，读取简单的路由文件，之后提示用户输入要检索的URL。

```
#include <lcthw/tstree.h>
#include <lcthw/bstrlib.h>

TSTree *add_route_data(TSTree *routes, bstring line)
{
    struct bstrList *data = bsplit(line, ' ');
    check(data->qty == 2, "Line '%s' does not have 2 columns",
            bdata(line));

    routes = TSTree_insert(routes,
            bdata(data->entry[0]), blength(data->entry[0]),
            bstrcpy(data->entry[1]));

    bstrListDestroy(data);

    return routes;

error:
    return NULL;
}

TSTree *load_routes(const char *file)
{
    TSTree *routes = NULL;
    bstring line = NULL;
    FILE *routes_map = NULL;

    routes_map = fopen(file, "r");
    check(routes_map != NULL, "Failed to open routes: %s", file)
;

    while((line = bgets((bNgetc)fgetc, routes_map, '\n')) != NULL
) {
        check(btrimws(line) == BSTR_OK, "Failed to trim line.");
        routes = add_route_data(routes, line);
        check(routes != NULL, "Failed to add route.");
```

```
        bdestroy(line);
    }

    fclose(routes_map);
    return routes;

error:
    if(routes_map) fclose(routes_map);
    if(line) bdestroy(line);

    return NULL;
}

bstring match_url(TSTree *routes, bstring url)
{
    bstring route = TSTree_search(routes, bdata(url), blength(ur
l));

    if(route == NULL) {
        printf("No exact match found, trying prefix.\n");
        route = TSTree_search_prefix(routes, bdata(url), blength
(url));
    }

    return route;
}

bstring read_line(const char *prompt)
{
    printf("%s", prompt);

    bstring result = bgets((bNgetc)fgetc, stdin, '\n');
    check_debug(result != NULL, "stdin closed.");

    check(btrimws(result) == BSTR_OK, "Failed to trim.");

    return result;

error:
    return NULL;
}

void bdestroy_cb(void *value, void *ignored)
{
    (void)ignored;
    bdestroy((bstring)value);
}

void destroy_routes(TSTree *routes)
{
    TSTree_traverse(routes, bdestroy_cb, NULL);
    TSTree_destroy(routes);
}
```

```c
int main(int argc, char *argv[])
{
    bstring url = NULL;
    bstring route = NULL;
    check(argc == 2, "USAGE: urlor <urlfile>");

    TSTree *routes = load_routes(argv[1]);
    check(routes != NULL, "Your route file has an error.");

    while(1) {
        url = read_line("URL> ");
        check_debug(url != NULL, "goodbye.");

        route = match_url(routes, url);

        if(route) {
            printf("MATCH: %s == %s\n", bdata(url), bdata(route)
);
        } else {
            printf("FAIL: %s\n", bdata(url));
        }

        bdestroy(url);
    }

    destroy_routes(routes);
    return 0;

error:
    destroy_routes(routes);
    return 1;
}
```

之后我创建了一个简单的文件，含有一些用于交互的伪造的路由：

```
/ MainApp /hello Hello /hello/ Hello /signup Signup /logout Logo
ut /album/ Album
```

你会看到什么

一旦你使 `urlor` 工作，并且创建了路由文件，你可以尝试这样：

```
$ ./bin/urlor urls.txt
URL> /
MATCH: / == MainApp
URL> /hello
MATCH: /hello == Hello
URL> /hello/zed
No exact match found, trying prefix.
MATCH: /hello/zed == Hello
URL> /album
No exact match found, trying prefix.
MATCH: /album == Album
URL> /album/12345
No exact match found, trying prefix.
MATCH: /album/12345 == Album
URL> asdfasfdasfd
No exact match found, trying prefix.
FAIL: asdfasfdasfd
URL> /asdfasdfasf
No exact match found, trying prefix.
MATCH: /asdfasdfasf == MainApp
URL>
$
```

你可以看到路由系统首先尝试精确匹配，之后如果找不到的话则会尝试前缀匹配。这主要是尝试这二者的不同。根据你的URL的语义，你可能想要之中精确匹配，始终前缀匹配，或者执行二者并选出“最好”的那个。

如何改进

URL非常古怪。因为人们想让它们神奇地处理它们的web应用所具有的，所有疯狂的事情，即使不是很合逻辑。在这个对如何将 `TSTree` 用作路由的简单演示中，它具有一些人们不想要的缺陷。比如，它会把 `/al` 匹配到 `Album` ，它是人们通常不想要的。它们想要 `/album/*` 匹配到 `Album` 以及 `/al` 匹配到404错误。

这并不难以实现，因为你可以修改前缀算法来以你想要的任何方式匹配。如果你修改了匹配算法，来寻找所有匹配的前缀，之后选出“最好”的那个，你就可以轻易做到它。这种情况下， `/al` 回匹配 `MainApp` 或者 `Album` 。获得这些结果之后，就可以执行一些逻辑来决定哪个“最好”。

另一件你能在真正的路由系统里做的事情，就是使用 `TSTree` 来寻找所有可能的匹配，但是这些匹配是需要检查的一些模式串。在许多web应用中，有一个正则表达式的列表，用于和每个请求的URL进行匹配。匹配所有这些正则表达式非常花时间，所以你可以使用 `TSTree` 来通过它们的前缀寻找所有可能的结果。于是你就可以缩小模式串的范围，更快速地做尝试。

使用这种方式，你的URL会精确匹配，因为你实际上运行了正则表达式，它们匹配起来更快，因为你通过可能的前缀来查找它们。

这种算法也可用于所有需要用户可视化的灵活路由机制。域名、IP地址、包注册器和目录，文件或者URL。

附加题

- 创建一个实际的引擎，使用`Handler`结构储存应用，而不是仅仅储存应用的字符串。这个结构储存它所绑定的URL，名称和任何需要构建实际路由系统的东西。
- 将URL映射到`.so`文件而不是任意的名字，并且使用`dlopen`系统动态加载处理器，并执行它们所包含的回调。将这些回调放进你的`Handler`结构体中，之后你就用C编写了动态回调处理器系统的全部。

"解构 K&R C" 已死

> 原文：Deconstructing K&RC Is Dead
>
> 译者：飞龙

我彻底失败了。我放弃了多年以来尝试理清C语言如何编写的想法，因为它的发明是有缺陷的。起初，我的书中有一章叫做"解构 K&R C"。这一章的目的是告诉人们永远不要假设它们的代码是正确的，或者对于任何人的代码，不管它有多出名，也不能避免缺陷。这看起来似乎并不是革命性的想法，并且对我来说它只是分析代码缺陷和编写更好更可靠代码的一部分。

多年以来，我在写这本书的这一块时收到重挫，并且收到了比任何其它事情更多的批评和侮辱。不仅如此，而且书中这部分的批评以这些话而结束，"你是对的，但是你认为他们的代码很烂这件事是错的。"我不能理解，有一群被认为很聪明的人，他们的大脑中充满理性，却坚持"我可以是错的，但是同时也可以是对的"的观点。我不得不与这些学究在C IRC channels、邮件列表、评论上斗争，这包括每一个它们提出一些怪异的、迂腐的刻薄意见的情况，需要我对我的文章进行更多的逻辑性修改来说服他们。

有趣的一点是，在我写这部分之前，我收到了本书许多正面的评论。当时本书还在写作中，所以我觉得确实需要改进。我甚至设置了一些奖金让人们帮助改进。但可悲的是，一旦他们被自己的英雄蒙蔽，所崇拜的基调就发生了翻天覆地的变化。我变得十分令人讨厌，只不过是尝试教人们如何安全使用一个极易出错的垃圾语言，比如C语言。这是我很擅长的东西。

这些批评者向我承认，他们不写C代码也不教授它，他们只是死记硬背标准库来"帮助"其它人，这对我来说并不重要。我以一个开放的心态试图解决问题，甚至设置奖金给那些有助于修复它的人，这也不重要。这可以使更多的人爱上C语言，并且使其它人入门编程，这更不重要。重要的是我"侮辱"了他们的英雄，这意味着我所说的话永远地完蛋了，没有人会再次相信我。

坦率地说，这是编程文化极为的黑暗、丑陋、邪恶的一面。他们整天在说，"我与你们同在"，但是如果你不屈服于大师们海量的学识，以及乞求他们准许你质疑他们所信奉的东西，你突然就会变成敌人。程序员费尽心机地把自己放在权力的宝座上，来要求别人赞许他们高超的记忆能力，或者对一些微不足道的琐事的熟知，并且会尽全力消灭那些胆敢质疑的人。

这非常恶心，我对此也没什么能做的。我对老程序员无能为力。但他们注定会失败。它们通过标准化记忆所积累的学识，也会在咸鱼的下一次翻身中蒸发掉。它们对考虑如何事物的运作方式，以及如何改进它们，或者将它们的手艺传授给他人毫无兴趣，除非这里面涉及到大量的阿谀奉承并让他们觉得很爽。老程序员总会完蛋的。

他们向现在的年轻程序员施压，我对此并不能做任何事情。我不能阻止无能程序员的诽谤，他们甚至根本不像专业的C程序员那样。然而，我宁愿使本书有助于那些想要学习C语言以及如何编写可靠的软件的人，而不是和那些思维闭锁的保守派做

斗争。它们贪图安逸的行为给人一种感觉，就是他们知道更多迂腐的、可怜的小话题，就比如未定义行为。

因此，我删除了书中的K&R C部分，并且找到了新的主题。我打算重写这本书，但是并不知道如何去做。我犹如在地狱中，因为我自己非常执着于我觉得很重要的一些事情，但我不知道如何推进。我现在算是明白了这是错的，因为它阻碍我将一些与C不相关的重要技巧教给许多新的程序员，包括编程规范、代码分析、缺陷和安全漏洞的检测，以及学习其它编程语言的方法。

现在我明白了，我将为这本书制作一些课程，关于编写最安全的C代码，以及将C语言代码打破为一种学习C和编程规范的方式。我会卑微地说我的书只是一个桥梁，所有人应该去读K&R C来迎合这些学究，并且在这些黄金法则的脚下顶礼膜拜。我要澄清我的C版本限制于一个固定的目的之中，因为这让我的代码更安全。我一定会提到所有迂腐的东西，比如每个书呆子式的，关于20世纪60年代的PDP-11电脑上空指针的要求。

之后，我会告诉人们不要再去写别的C程序。这不会很明显，完全不会，但我的目标是将人们从C带到能更好地编程的其它语言中。Go、Rust或者Swift，是我能想到的能处理C语言主要任务新型语言，所以我推荐人们学习它们。我会告诉他们，他们的技能在于发现缺陷，并且对C代码的严格分析将会对所有语言都有巨大的好处，以及使其它语言更易于学习。

但是C呢？C已经死了，它是为想要争论A.6.2章第四段的指针未定义行为的老程序员准备的。谢天谢地，我打算去学习Go（或者Rust，或者Swift，或者其它任何东西）了。

捐赠名单

感谢以下童鞋的捐助，你们的慷慨是我继续的动力：

donor	value
jxdwinter	6.00
贾**@悠云.com	20.00
Mr.Moon	2.00